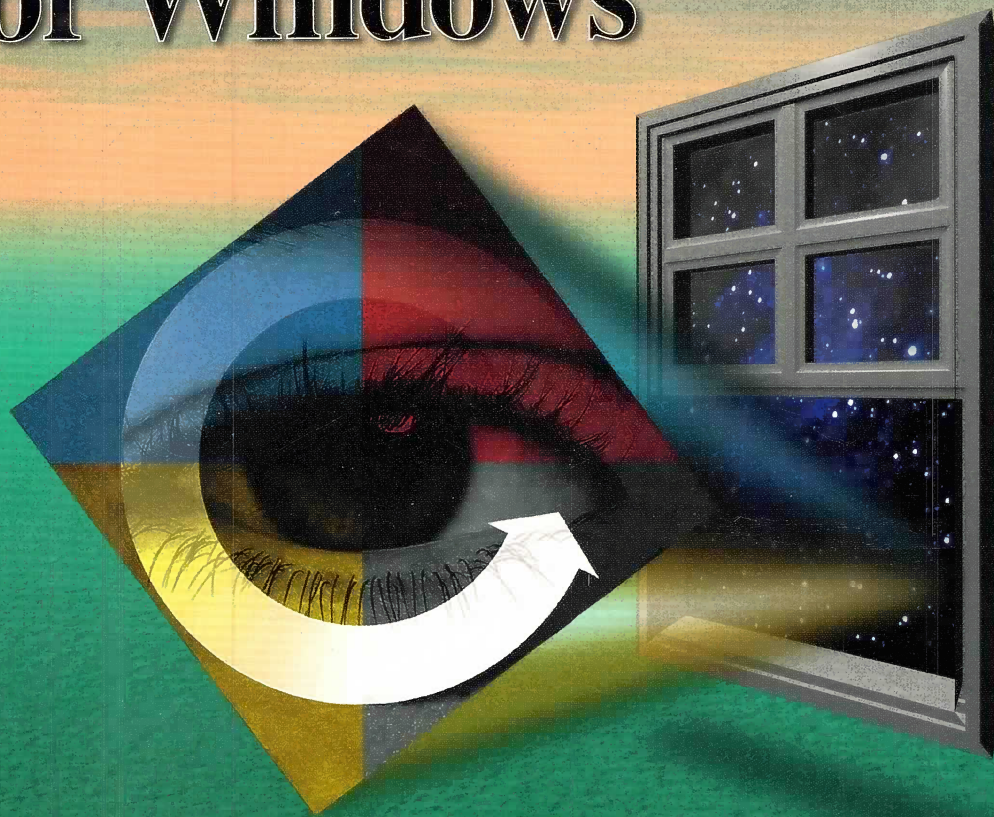


# Programming with VisualAge for C++ for Windows



- ◆ Free Evaluation Copy of VisualAge for C++ for Windows
- ◆ Free Evaluation Copy of DB2 for Windows NT
- ◆ Running Sample Application with Source Files

**IBM**

**Marc Carrel-Billiard  
Michael Friess  
Isabelle Mauny**



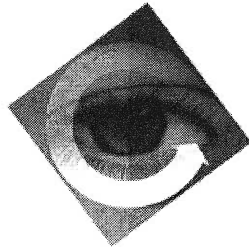
**International Technical Support Organization**





# **Programming with VisualAge for C++ for Windows**

# The VisualAge Series



Bitterer, Brassard, Nadal, and Wong

*VisualAge and Transaction Processing in a Client/Server Environment*

Bitterer, Hamada, Oosthuizen, Porciello, and Rambek

*AS/400 Application Development with VisualAge for Smalltalk*

Bitterer, Carrel-Billiard, Bianco, Bosman-Clark, Georges, and Tsang

*World Wide Web Server Development with VisualAge for C++ and Smalltalk*

Carrel-Billiard, Jakab, Mauny, and Vetter

*Object-Oriented Application Development with VisualAge for C++ for OS/2*

Carrel-Billiard, Friess, and Mauny

*Programming with VisualAge for C++ for Windows*

Fang, Chu, and Weyerhäuser

*VisualAge for Smalltalk and SOMobjects: Developing Distributed Object Applications*

Fang, Guyet, Haven, Vilmi, and Eckmann

*VisualAge for Smalltalk Distributed: Developing Distributed Object Applications*



# Programming with **VisualAge for C++** for Windows

**Marc Carrel-Billiard**  
**Michael Friess**  
**Isabelle Mauny**



INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION  
SAN JOSE, CALIFORNIA 95120



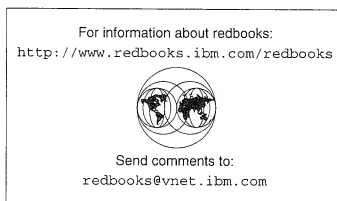
PRENTICE HALL PTR  
UPPER SADDLE RIVER, NEW JERSEY 07458

© Copyright International Business Machines Corporation 1995, 1997. All rights reserved.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

This edition applies to Version 3.5 of the VisualAge for C++ for Windows product set, and to all subsequent releases and modifications until otherwise indicated in new editions.

Comments about ITSO Technical Bulletins may be addressed to:  
IBM Corporation ITSO, 471/80-E2, 650 Harry Road, San Jose, California 95120-6099



Published by Prentice Hall PTR  
Prentice-Hall, Inc.  
A Simon & Schuster Company  
Upper Saddle River, NJ 07458

*Acquisitions Editor:* Michael E. Meehan

*Manufacturing Manager:* Alexis R. Heydt

*Cover Design:* John Fitzgerald

*Cover Design Director:* Jerry Votta

*Copy Editors:* Shirley Hentzell, Maggie Cutler

*Production Supervision:* Craig Little

The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact:  
Corporate Sales Department, Prentice Hall PTR, One Lake Street, Upper Saddle River, NJ 07458  
Phone: 800-382-3419; FAX: 201-236-7141; E-mail (Internet): [corpsales@prenhall.com](mailto:corpsales@prenhall.com)

Printed in the United States of America

1 0 9 8 7 6 5 4 3 2 1

ISBN 0-13-618208-9

Prentice-Hall International (UK) Limited, *London*

Prentice-Hall of Australia Pty. Limited, *Sydney*

Prentice-Hall Canada Inc., *Toronto*

Prentice-Hall Hispanoamericana, S.A., *Mexico*

Prentice-Hall of India Private Limited, *New Delhi*

Prentice-Hall of Japan, Inc., *Tokyo*

Simon & Schuster Asia Pte. Ltd., *Singapore*

Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*



To my buddies from the poor and rich sides of Normington Way, even with your outstanding way of playing tennis doubles, you made one frenchy a Californian at heart. To my brothers and my sister for teasing their “besogneux” computer nerd. To my Fan club of l’Ile-Tudy and their unforgettable mackerel fishing parties!

Marc

To my girlfriend and my family for the home and love they give to me. To Isabelle, Marc, Rainer and all other colleagues who make work in IBM something special. To all people joining me on my way with their patience, charity, vision, creativity, wisdom, trust, joy, power and love.

Michael

To my grandma, that never understood why I had to travel to California to work on a book, and to my best friend Joel, for always supporting me.

Isabelle





---

# Contents

<b>Special Notices</b> .....	xxi
<b>Foreword</b> .....	xxv
<b>Preface</b> .....	xxvii
What Makes This Book Different .....	xxvii
How This Book Is Organized .....	xxvii
Special Conventions in This Book .....	xxx
Related Publications .....	xxx
International Technical Support Organization Publications .....	xxxii
ITSO on the Internet .....	xxxiii
VisualAge for C++ Support .....	xxxiv
About the Authors .....	xxxv
Acknowledgments .....	xxxv

---

## **Part 1. Introduction to the VisualAge for C++ Environment** .....

### **Chapter 1. VisualAge for C++ and Application Development** .....

Visual Programming .....	4
Object Talk .....	6
Objects .....	6
Classes .....	7
Inheritance .....	8
Encapsulation .....	10
Polymorphism .....	10
Object-Oriented Methods .....	11
Visual Modeling Technique .....	13
Analysis .....	14
Design .....	15
Implementation .....	15
Visual Programming with VisualAge for C++ .....	16

### **Chapter 2. Getting Started in a VisualAge for C++ Environment** .....

A Short Overview .....	20
Managing Your Project .....	20
WorkFrame Concepts .....	20
Creating a Project with WorkFrame .....	22
Creating Composite Projects .....	23
The MakeMake and Build Facilities .....	23
Customizing a Project with Build Smarts .....	24
Generating Your Code .....	24
Using Visual Builder .....	24
Accessing DB2 Tables with Data Access Builder .....	33

---

Building from Blocks .....	35
The Resource Workshop .....	39
Building Your Application .....	39
Editing Your Code .....	40
Compiling .....	41
Linking .....	45
Understanding Your Code .....	46
Browsing Your C++ Hierarchy .....	46
Debugging Your Code .....	50
Performance Analysis .....	52

---

<b>Part 2. Developing with VisualAge for C++ .....</b>	<b>57</b>
--	-----------

<b>Chapter 3. Analysts at Work .....</b>	<b>59</b>
--	-----------

Collecting the Material .....	60
Problem Domain .....	61
Requirement Specifications .....	62
Thread and Subplots .....	65
Use Case Model .....	66
User Interface Prototype .....	68
Defining Roles .....	69
Patterns and Types .....	70
Finding Objects .....	71
Class Dictionary and CRC Cards .....	72
Defining Interactions and Relations .....	74
Defining Contexts .....	79

<b>Chapter 4. Designers at Work .....</b>	<b>81</b>
---	-----------

System Design .....	83
Partition Object Model into Subsystems .....	83
Map Subsystems to VisualAge for C++ Subapplications .....	84
Select the Implementing Platform .....	85
Define Data Placement and Data Processing .....	86
Refine Contexts .....	87
Object Design .....	87
Design the Solution Domain Classes .....	87
Design the Nonvisual Parts .....	88
Design the GUI with the Visual Parts .....	89
Design the Persistent Data .....	90
Refining the Design Model .....	90
Refining the Property Retrieval Scenario .....	93
Refining the Property Creation Scenario .....	96
Refining the Property Update Scenario .....	99
Refining Roles .....	100



<b>Part 3. Building the Visual Realty Application</b> . . . . .	103
<b>Chapter 5. Setting Up the Development Environment</b> . . . . .	105
WorkFrame Projects Organization . . . . .	106
File Organization . . . . .	107
Associating IWP Files with WorkFrame . . . . .	109
Creating and Customizing the Dacslib Project . . . . .	110
Creating the Dacslib Project . . . . .	110
Customizing the Dacslib Project . . . . .	112
Creating and Customizing the Visual Realty Projects . . . . .	115
Creating the Visual Realty Main and Subsystem Projects . . . . .	115
Creating the Help Project . . . . .	116
Customizing the Visual Realty Main and Subsystem Projects . . . . .	117
Naming Conventions . . . . .	122
Run-Time Considerations . . . . .	123
Modifying the WorkFrame Configuration File . . . . .	123
Customization Possibilities . . . . .	124
<b>Chapter 6. Mapping Relational Tables Using Data Access Builder</b> . . . . .	131
Defining the Tables and Views . . . . .	132
Mapping Tables to Parts . . . . .	133
Parts Produced . . . . .	139
Using Data Access Builder Parts with Visual Builder . . . . .	140
<b>Chapter 7. Creating Visual Parts</b> . . . . .	147
AAddressView . . . . .	152
Setting the Tabbing and Depth Order . . . . .	157
Promoting a Part Feature . . . . .	158
APropertyView . . . . .	160
Using a Notebook Control . . . . .	162
Building the Pages of a Notebook . . . . .	163
APropertyCreateView . . . . .	178
APropertyUpdateView . . . . .	180
ADeleteDialogView . . . . .	183
ASimulMortgageView . . . . .	187
ASimulMortgageFrameView . . . . .	191
APropertySearchResultView . . . . .	192
Using a Container . . . . .	192
Adding Columns to a Container . . . . .	197
APropertySearchParameterView . . . . .	200
Using Check Box Control . . . . .	200
Using Collection Combination-Box Control . . . . .	201
AUploadView . . . . .	208
APropertyManagementView . . . . .	211
Using Graphic Push Buttons . . . . .	212
ALogonView . . . . .	215
ARealSettingsView . . . . .	218

ARealMainView .....	220
<b>Chapter 8. Creating Nonvisual Parts .....</b>	<b>225</b>
AMarketingInfo .....	226
Defining the Class Interface .....	227
Refining the Feature Source Code .....	230
AMortgageCalculator .....	233
Defining the Class Interface .....	234
Writing the Source Code .....	235
Event Handler .....	237
Writing the Code for Your Event Handler Class .....	238
Creating a Class Interface Part from Your Event Handler Class .....	243
Using Your Keyboard Handler .....	244
<b>Chapter 9. Connecting the Parts .....</b>	<b>247</b>
AAddressView .....	248
Using Sample Parts .....	248
Connecting a Nonvisual Part to a Visual Part .....	249
Passing a Parameter to a Connection Dynamically .....	250
APropertyView .....	251
Selecting a Video File .....	251
Adding Multimedia Features .....	253
Adding a Pop-up Menu .....	254
Connecting AMarketingInfo to the Marketing Page .....	256
Design Considerations .....	258
APropertyCreateView .....	259
Using Variable Parts .....	266
Managing the Database Connection .....	269
Adding Fly-over Help to a Control .....	270
Passing a Parameter to a Connection Staticly .....	271
Using Custom Logic .....	271
APropertyUpdateView .....	274
Showing Exception in a Message Box .....	282
Using a Member Function Connection .....	283
Updating a Window Title Dynamically .....	286
ADeleteDialogView .....	286
ASimulMortgageView .....	287
ASimulMortgageFrameView .....	289
APropertySearchResultView .....	289
Selecting Properties from the Database .....	291
Retrieving Information Across Multiple Tables .....	292
Using an Object Factory to Update the Database .....	295
Deleting a Property .....	298
APropertyDelete .....	302
Using the Composition Editor to Build a Nonvisual Part .....	302
APropertySearchParameterView .....	307
Managing the User Input .....	308
Adding The Mortgage Calculation .....	310

---

Building the Clause . . . . .	311
Using a Message Box to Display the Clause . . . . .	318
AUploadView . . . . .	324
APropertyManagementView . . . . .	328
ALogonView . . . . .	331
ARealSettingsView . . . . .	331
ARealMainView . . . . .	336
DB2 for Windows Authentication . . . . .	336
Logging on to the Database . . . . .	337
Accessing the Application Settings and the Property Subsystem . . . . .	340
Tearing Off an Attribute . . . . .	344
Adding Help to Your Application . . . . .	346

---

<b>Part 4. If You Want to Know More...</b> . . . . .	351
--	-----

<b>Chapter 10. More about Visual Builder...</b> . . . . .	353
---	-----

Notification Framework Concepts . . . . .	353
How Visual Builder Uses the Notification Framework . . . . .	354
Using Connections as Notifiers . . . . .	361
From Classes to Nonvisual Parts in Visual Builder . . . . .	363
Protecting your Code . . . . .	364
Creating the FlatFile Nonvisual Part . . . . .	365
Describing the Part Interface . . . . .	366
Modifying Your Code . . . . .	369
When Parts Become Observers... . . . .	370

<b>Chapter 11. More about Data Access Builder...</b> . . . . .	373
--	-----

Accessing the Database from a Windows 95 Client . . . . .	374
Setting Up the Database Server . . . . .	375
Setting Up the Windows 95 Client . . . . .	380
Impact on the Application Development . . . . .	381
A Short History of SQL Data Access . . . . .	382
Embedded SQL . . . . .	383
Call Level Interface . . . . .	385
Using ODBC as Data Access Method . . . . .	389
Differences in the Data Access Classes . . . . .	390
Impacts on the Application . . . . .	391
Building the Data Access Classes . . . . .	393
Rebuilding the Whole Application . . . . .	394
Implementing the Database as dBase Files . . . . .	397
A Small Introduction to DBF Files . . . . .	397
Representing Databases with DBF Files . . . . .	398
Impacts on the Application . . . . .	401
Creating the DBF Files . . . . .	402
Registering the ODBC Data Source . . . . .	404
Generating the Data Access Classes . . . . .	406
Creating Classes for Database Views . . . . .	408

---

Building the Application .....	412
Enhancing the Data Access in the Application .....	413
Ordering Data .....	413
Buffering Data .....	414
Displaying Data .....	414
Adding Calculated Attributes .....	418
<b>Chapter 12. More about SOM...</b> .....	421
SOM: A Complement to C++ .....	422
Release-to-Release Binary Compatibility .....	423
Language Neutrality .....	424
The SOM Technology .....	425
The SOM Run-Time Environment .....	426
SOM by Example .....	427
Defining the Interface of the Calculator Class .....	427
Generating C++ Bindings with the SOM Compiler .....	428
Completing the Class Implementation .....	429
Writing a Client Application .....	430
Creating a SOM DLL .....	432
Modifying the SOM DLL .....	436
Advanced Features .....	437
SOM Metaclasses .....	437
Functions Overriding .....	441
Attributes vs. Instance Variables .....	442
Understanding Direct-to-SOM .....	443
Restrictions Imposed by DTS .....	444
DTS by Example .....	444
Controlling the Compiler Output with Pragas .....	446
<b>Chapter 13. More about CDF...</b> .....	453
Introducing the Compound Document Technology .....	454
Object Linking and Embedding .....	455
The Compound Document Framework .....	457
Models and Views .....	458
Building a Component Server .....	464
Creating Your Project with WorkFrame .....	464
Creating a Server Model .....	467
Creating a Server View .....	473
Handling Communications Between Model and View .....	486
Instantiating the Stationery and Creating the Main Function .....	489
Compiling and Linking Your Application .....	489
Using Your OLE Server .....	490
Developing OLE Components with the Visual Builder .....	491
Creating Your Project with WorkFrame .....	492
Modifying the Visual Part .....	493
Modifying the Class Interface Part .....	495
Modifying the Server Model Code .....	496
Modifying the Server View Code .....	500



---

<b>Appendix A. Installing the Application</b> .....	509
<b>Appendix B. OMT Notation</b> .....	511
<b>Appendix C. Database Definition</b> .....	515
<b>Appendix D. Class Dictionary</b> .....	521
Visual Parts .....	522
Nonvisual Parts .....	522
<b>Appendix E. Source Listings</b> .....	525
BuildClause member function .....	526
Flat File Class .....	526
<b>Glossary</b> .....	535
<b>List of Abbreviations</b> .....	547
<b>Index</b> .....	549

---

---

## Figures

1. Inheritance .....	9
2. VMT: A Complementary Approach to Object-Orientation .....	14
3. Interface to Alter Options for the VisualAge for C++ Compiler .....	21
4. Project Smarts Catalog View: Data Access Application .....	22
5. Example of Composite Project .....	23
6. Primitive and Composite Parts .....	25
7. Sample Part Interface: SmartHouse .....	26
8. Sample Connections: SmartHouse Monitoring Window .....	27
9. Visual Builder: Composition Editor .....	29
10. Part Interface Editor: Attribute Creation .....	30
11. defaultButtonsPanel Composite Part .....	31
12. Visual Builder: Class Editor .....	32
13. Database Access: From Mapping to Parts Generation .....	34
14. User Interface Class Library Architecture .....	36
15. LPEX: Source Formatting and Dynamic Error Detection .....	40
16. Language-Independent Implementation with SOM .....	44
17. Browser List Window: List Members with Inheritance .....	47
18. Browser Graph Window: Graph All Callers and Callees .....	48
19. Browser Graph Window: Graph All Includers .....	49
20. Visual Builder: Creating an Event-to-Member Connection .....	50
21. Breakpoint List Window .....	51
22. Call Nesting Diagram Window .....	54
23. Dynamic Call Graph Window: Nodes Of Functions .....	55
24. Statistics Window .....	55
25. Use Case Representation .....	66
26. Visual Realty Use Cases .....	68
27. User Interface Samples .....	69
28. Event-Trace Diagram for the Record Property Use Case .....	76
29. State Transition Diagram of Property Status .....	77
30. Analysis Object Model of the Visual Realty Application .....	78
31. From Analysis to Design .....	82
32. Visual Realty System Platform .....	86
33. Design Model: Reveal Hidden Objects .....	88
34. Design Object Model of the Property Subsystem: First Cut .....	92
35. Event-Trace Diagram for the Property Search Use Case .....	94
36. Design Object Model of the Property Subsystem: Second Cut .....	95
37. Event-Trace Diagram for the Property Creation Use Case .....	97
38. Design Object Model of the Property Subsystem: Third Cut .....	98
39. Event-Trace Diagram for the Property Update Use Case .....	99
40. Design Object Model of the Property Subsystem: Fourth Cut .....	100
41. Project Organization for the Visual Realty Application .....	107
42. File Organization for the Visual Realty Application .....	108
43. Location Settings for the Dacslib Project .....	111
44. Data Access Builder Create Classes Window .....	134
45. Data Access Builder Main Window .....	136
46. Property Settings Window .....	137
47. Data Access Builder Attributes Page of the Settings Window .....	138

---

48. Simple Application with Data Access Builder .....	143
49. Visual Realty Application in Action .....	148
50. View Hierarchy .....	149
51. AAddressView Part .....	152
52. AAddressView Parts List .....	157
53. Promote EntryFieldStreetText Attribute of AAddressView .....	159
54. Windows Version of APropertyView Part .....	160
55. CUA Version of APropertyView Part .....	161
56. Windows Version Notebook for APropertyView .....	162
57. Characteristics Page Using a Viewport .....	165
58. Event Handler List Box .....	165
59. Address Page .....	169
60. Description Page .....	170
61. Video Page .....	171
62. Marketing Page .....	174
63. APropertyCreateView .....	178
64. APropertyUpdateView .....	181
65. ADeleteDialogView .....	183
66. ASimulMortgageView .....	188
67. ASimulMortgageFrameView .....	191
68. APropertySearchResultView .....	193
69. Container General Settings Page .....	197
70. Container Column General Settings Page .....	198
71. APropertySearchParameterView .....	201
72. IStringGeneratorForPropertyFn Declaration .....	204
73. AUploadView .....	209
74. APropertyManagementView .....	211
75. IGraphicPushButton General Settings Page .....	212
76. ALogonView .....	215
77. ARealSettingsView .....	218
78. ARealMainView .....	221
79. Creating AMarketingInfo Nonvisual Part .....	228
80. MarketingInfo Source Code Detail .....	231
81. Creating AMortgageCalculator Class Interface .....	234
82. AMortgageCalculator Header File: vrssimc.hpp .....	235
83. AMortgageCalculator Implementation File: vrssimc.cpp .....	236
84. NumOnlyKbdHandler and NumDecOnlyKbdHandler Header File .....	239
85. NumOnlyKBDHandler Definition .....	240
86. NumDecOnlyKbdHandler Definition .....	242
87. Visual Builder: Importing kbdhdr.vbe Part Information File .....	244
88. Simple Application with Handler .....	245
89. AAddressView Connections .....	249
90. Connections for Selecting a Video File .....	252
91. Building a Pop-up Menu for the Multiple-Line Edit Control .....	255
92. Connections between AMarketingInfo and Marketing Page .....	257
93. APropertyCreateView and Its Subparts .....	260
94. Attribute-to-Attribute Connections in APropertyCreateView .....	262
95. Event-to-Action Connections in APropertyCreateView .....	264
96. Connection Order for the Create Push Button .....	266

97. Simple View to Display Property Information .....	267
98. Reusing the Property View from Another Part: First Try .....	268
99. Simple Property View with Its Associated Variables .....	268
100. Reusing the Property View from Another Part: Second Try .....	269
101. Attribute-to-Attribute Connections in APropertyUpdateView. ....	275
102. Event-to-Action Connections in APropertyUpdateView .....	279
103. Connection Order for the Update Push Button .....	281
104. DaysOnMarket Public Method Declaration .....	283
105. DaysOnMarket Public Method Definition .....	284
106. Member Function Dialog Box .....	285
107. Connecting AMortgageCalculator with ASimulMortgageView .....	288
108. Querying the Database .....	291
109. Retrieving Information from Multiple Tables .....	293
110. Updating a Property .....	296
111. Deleting a Property .....	299
112. Building APropertyDelete .....	302
113. Order of Connections for APropertyDelete .....	305
114. Detail of the Class Editor .....	306
115. Code to Generate deleteEventId .....	306
116. Using APropertyDelete Part .....	307
117. Number of Bedrooms Selection .....	309
118. Activating The Mortgage Calculation .....	310
119. Code for Activating the Mortgage Simulation .....	311
120. Code Fragment of BuildClause Member Function .....	312
121. Subparts of APropertySearchParameterView .....	313
122. APropertySearchParameterView: The Big Picture .....	315
123. Using a Message Box to Display the Clause .....	319
124. Message Box Parameter .....	320
125. Using Message Box to Display a Warning Message .....	321
126. Message Box Displaying a Warning or Information Message .....	323
127. AUploadView .....	325
128. Browsing the IProfile Part's Features .....	328
129. APropertyManagementView .....	329
130. Structure of the Registry .....	332
131. ARealSettings Part .....	333
132. Logon to the Database .....	338
133. Application Settings and Property Subsystem Access .....	341
134. Tearing Off an Attribute .....	345
135. Adding Help to the Application .....	348
136. IVBConnection Class and its Derived Classes .....	355
137. Sample Window: Using an Event-to-Action Connection .....	356
138. Visual Builder Parts Initialization Process .....	358
139. Visual Builder Notification Flow .....	360
140. Sample2 Window: Using a Message Box for Exception Handling .....	361
141. Part-New Window: Creating a FlatFile Nonvisual Part .....	365
142. Part Interface Editor Window: Creating an Attribute Definition .....	367
143. The defaultButtons Composite Visual Part .....	370
144. DB2 Platforms .....	374
145. Configuring the NetBIOS Interface .....	378



---

146. The ODBC Components .....	386
147. Data Sources Window .....	390
148. Default Datastore Setting Code .....	393
149. Changing ARealMain part .....	395
150. Registering dbf Files as an ODBC Data Source .....	405
151. Associating mdx and dbf Files .....	405
152. Data Definition for Dummy Tables .....	409
153. Changed Select Statements in PropAdLg.cpp .....	411
154. Overriding the asString Method .....	416
155. Overriding the forDisplay Method .....	418
156. Derived Property_view class Propview.hpp .....	419
157. Impact of Changes in a C++ Library .....	423
158. SOM Classes and Metaclasses Relationship .....	426
159. SOMClass, SOM Object, and SOMClassMgr Relationship .....	427
160. In-place Activation of a Calculator Server .....	455
161. Model and View of an OLE Application .....	458
162. Component Embedded within a Container Embedder Model .....	459
163. Key Objects of an Embedding Relationship .....	460
164. The Calculator Component Server .....	464
165. Calculator Server Project .....	466
166. STRINGTABLE Window .....	478
167. Addition Event Trace Diagram .....	486
168. Out-Place Activation of the Calculator Server .....	491
169. Resource Workshop Window .....	501
170. OMT Notation: Object Model .....	512
171. OMT Notation: State Diagram .....	513
172. BuildClause Member Function: Declaration .....	526
173. Flat File Class: H File .....	526
174. Flat File Class: HPP File .....	527
175. Flat File Class: CPP File .....	529
176. Flat File Class: HPV File .....	531
177. Flat File Class: CPV File .....	532

## Tables

1. SmartHouse Connections . . . . .	28
2. Extended CRC Cards for Buyer . . . . .	73
3. Extended CRC Cards for Property . . . . .	74
4. Extended CRC Cards for Sale Transaction . . . . .	74
5. Deliverables of Analysis and Design . . . . .	82
6. Relational Table Identifiers . . . . .	138
7. Constructing AAddressView Part . . . . .	155
8. Building APropertyView As a Notebook . . . . .	163
9. Building the Characteristics Page . . . . .	166
10. Building the Address Page . . . . .	169
11. Building the Description Page . . . . .	170
12. Building the Video Page . . . . .	172
13. Building the Marketing Page . . . . .	175
14. Promoted Features of APropertyView . . . . .	176
15. Constructing APropertyCreateView Part . . . . .	178
16. Constructing APropertyUpdateView Part . . . . .	181
17. Building ADeleteDialogView . . . . .	185
18. Building ASimulMortgageView . . . . .	188
19. Building ASimulMortgageFrameView . . . . .	191
20. Building APropertySearchResultView: Building a Container . . . . .	194
21. Building APropertySearchResultView: Adding Container Columns . . . . .	198
22. Building APropertySearchParameterView . . . . .	204
23. Building AUploadView . . . . .	209
24. Building APropertyManagementView . . . . .	213
25. Building ALogonView . . . . .	215
26. Building ARealSettingsView . . . . .	219
27. Building ARealMainView . . . . .	221
28. AMarketingInfo Attributes . . . . .	228
29. AMortgageCalculator Attributes . . . . .	233
30. AMortgageCalculator Function . . . . .	233
31. Implementing a State Selection . . . . .	249
32. Implementing a Video File Selection . . . . .	252
33. Implementing Multimedia Features . . . . .	253
34. Building a Pop-up Menu . . . . .	255
35. Using AMarketingInfo Part . . . . .	257
36. Adding Parts in APropertyCreateView . . . . .	260
37. Making Attribute-to-Attribute Connections in APropertyCreateView . . . . .	262
38. Making Event-to-Action Connections in APropertyCreateView . . . . .	265
39. Adding Subparts in APropertyUpdateView . . . . .	275
40. Making Attribute-to-Attribute Connections in APropertyUpdateView . . . . .	277
41. Making Event-to-Action Connections in APropertyUpdateView . . . . .	280
42. Connecting AMortgageCalculator with ASimulMortgageView . . . . .	288
43. Adding Parts to Query the Database . . . . .	292
44. Adding Parts to Retrieve Information from Multiple Tables . . . . .	293
45. Connecting Parts to Retrieve Property Information . . . . .	294
46. Updating the Database . . . . .	297
47. Connecting Parts to Update Property Information . . . . .	297
48. Adding Parts to Delete a Property from the Database . . . . .	299
49. Connecting Parts to Delete a Property from the Database . . . . .	300
50. Adding Parts to Build APropertyDelete . . . . .	302

---

51.	Connecting Parts to Build APropertyDelete . . . . .	304
52.	Adding Parts to Build APropertySearchParameterView . . . . .	313
53.	Connecting Parts to Build APropertySearchParameterView . . . . .	315
54.	Using a Message Box to Display the Clause . . . . .	319
55.	Using Message Box to Display a Warning Message . . . . .	321
56.	Using Message Box to Display a Warning or Information Message . . . . .	323
57.	Building AUploadView Part . . . . .	325
58.	Building APropertyManagementView Part . . . . .	329
59.	Building ARealSettingsView Part . . . . .	333
60.	Adding Parts for the Logon Function . . . . .	338
61.	Connecting Parts for the Logon Function . . . . .	339
62.	Adding Parts to Access the Application Settings and Property Subsystem . . . . .	341
63.	Connecting Parts for Property Subsystem and Settings Access . . . . .	343
64.	Adding Help to the Application . . . . .	348
65.	Adding Help Resource Numbers for ARealMainView . . . . .	349
66.	Adding Help Resource Numbers for ARealSettingsView . . . . .	350
67.	Comparing Static and Dynamic SQL . . . . .	384
68.	Changing ARealMain for ODBC Data Access Method . . . . .	395
69.	Data Types of dBase IV DBF Files . . . . .	398
70.	Data Types of ODBC dBase Driver . . . . .	400
71.	Shortened Table and Attribute Names of REAL dbf Files . . . . .	407
72.	Shortened Table and Attribute Names of REAL View dbf Files . . . . .	410
73.	String Table Resources for CalcServerView . . . . .	478
74.	Resource Identifiers for CalcServerView . . . . .	479
75.	File List for the SimulMortgage OLE Component Project . . . . .	493
76.	File List for the SimulMortgage OLE Component Project . . . . .	495
77.	Database Layout for the REAL Database . . . . .	516

---

## Special Notices

This publication is intended to help project leaders to better understand the VisualAge for C++ environment. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge for C++. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge for C++ for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service.

Information in this book was developed in conjunction with use of the equipment specified and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (VENDOR) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

---

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX®	Common User Access
CSet ++™	DB2™
CUA™	DB2/2™
DRDA™	MVS/ESA®
Presentation Manager™	IBM®
Multimedia Presentation Manager/2™	OS/2®
SOMobjects™	OS/2 Warp®
OS/400®	VisualAge™
Workplace Shell™	WorkFrame/2™
WebExplorer™	

The following terms are trademarks of other companies:

Borland® and dBase® are registered trademark of Borland International, Inc.

C-bus™ is a trademark of Collary, Inc.

HP-UX™ are trademarks of Hewlett-Packard Company.

i386™ and Pentium™ are trademarks of Intel Corporation.

Informix® is a registered trademark of Informix Software, Inc.

Ingres™ is a trademark of Ingres Corporation.

INTERSOLV™, Datadirect are trademarks of INTERSOLV Corporation.

IPX/SPX® are registered trademarks of Novell, Inc.

Lotus®, Approach® are registered trademarks of Lotus Development Corporation.

---

Microsoft®, MS®, Access®, Excel® are registered trademarks and ODBC™, Windows™, Windows™ NT are trademarks of Microsoft Corporation.

Motif® is a registered trademark of Open Software Foundation.

Oracle® is a registered trademark of Oracle Corporation.

PC Direct™ is a trademark of Ziff Communications Company and is used by IBM Corporation under licence.

Sinix™ is a trademark of Siemens.

Smalltalk™ is a trademark of Xerox Corporation.

Solaris® is registered trademark of Sun Microsystems.

SQLBase® is a registered trademark of Gupta Technologies.

SYBASE® is a registered trademark of Sybase, Inc.

UNIX® is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open™ is a trademark of X/Open Company Limited.

Other trademarks are trademarks of their respective companies.

The icons used in this book are from the ClipArt Collection of the CorelDRAW! Version 5 CDROM.

Some videos provided with the sample application are extracted from the CDROM Nitro Explosive Animationen © 1994 Data Becker.

---



---

## Foreword

When IBM set out to create a VisualAge™ for C++ product a few years ago, we had a couple of objectives. We had already created a very successful C++ IDE based product on OS/2 and AIX called CSET++, and were moving it to multiple platforms. We had also created the VisualAge Smalltalk product, with its tightly integrated visual development environment. The natural question from C++ programmers was “why not VisualAge for C++ as well”. This lead us to our first VisualAge for C++ on OS/2 in 1995, followed by this product for Win32 in 1996. It was designed to bring the powerful “construction from parts” metaphor to C++, along with a powerful GUI builder and the richest set of class libraries in the industry.

The Win32 version of this product represents the latest evolution in this technology, with the addition of new frameworks from Taligent that simplify the creation of OLE objects, and provide a higher level of abstraction that will support OpenDoc™ and other component architectures. We included data access frameworks and tools that simplify the mapping of relational structures to Objects, and have significantly reworked and enhanced major areas of the product. Above all, VisualAge for C++ for Windows™ demonstrates IBM's ongoing commitment to support cross-platform application development. applications built using VisualAge for C++ and its Open Class libraries can be made more portable to more platforms than with any other C++ in the industry.

I hope you enjoy this book, and using the copy of VA C++ that come with it.

Welcome to the Visual Age of programming!

John Swainson

Vice President Application Development Solutions  
Director IBM Toronto Lab.

---

---

## Preface

Welcome to the world of visual programming! With VisualAge for C++ for Windows you are ready to take the plunge into a radically new trend of programming. If you have just bought your IBM VisualAge for C++ and you are dying to build your first serious application, you are reading the right book. Indeed, learning VisualAge for C++ by example is all this book is about. With VisualAge for C++, application construction has never been easier. Even the most complex applications can be constructed from the large set of predefined parts from IBM Open class. This book will show you how you can employ IBM VisualAge for C++ for Windows, Version 3.5, to implement software systems that have been analyzed and designed by use of object-oriented methods. It introduces the Visual Modeling Technique, a complementary approach of existing object-oriented development techniques and illustrates how this approach is applied to build a real application featuring relational database support, video and vivid sound capacity, and numerous graphical controls for a truly intuitive graphical user interface.

## What Makes This Book Different

This book explains how to develop an application from the requirements specifications up to its coding with VisualAge for C++. Throughout the different chapters, you will be guided to develop your static and dynamic object models, using the Visual Modeling Technique. Then, you will translate your models visually in Visual Builder and generate their code automatically. This book is neither just a book on methodology nor just a book on programming: it is both!

For the first time, a book takes you by hand to roll out a complete application development cycle. So put on your cap of analyst-designer-developer and get ready for a trip to the visual programming world!

## How This Book Is Organized



This book consists of four parts. The first part (Chapters 1 and 2) introduces concepts and terms that go with visual programming and object-orientation and gives a first insight into the VisualAge for C++ development environment. In the second part (Chapters 3 and 4), we present the sample application that you will build in the last part. This part is devoted to analyzing and designing the static and dynamic model of the application to ease its implementation with VisualAge for C++. The third part (Chapters 5, 6, 7, 8, and 9) makes up the majority of the book, teaching you how to use VisualAge for

C++ and its versatile tools to develop the sample application from the ground up. The fourth part (Chapters 10, 11, 12 and 13) introduces more advanced features such as the IBM notification framework, the Direct-to-SOM support and the Compound Document Framework.

❑ **Chapter 1, “VisualAge for C++ and Application Development,” on page 3**

The first chapter welcomes you to the visual age of application development. You learn something about the new trends of software construction that have emerged during the past few years and how VisualAge for C++ meets these new challenges.

❑ **Chapter 2, “Getting Started in a VisualAge for C++ Environment,” on page 19**

The second chapter provides an overview of all of the tools and features that are part of the VisualAge for C++ package. We do not intend to replace the user guides, but we want to give you the keys that let you start off applying the tools.

❑ **Chapter 3, “Analysts at Work,” on page 59**

This chapter and the next one invite you to play the role of a novelist. We compare the analysis and design phases that precede the implementation of a successful and neatly structured software system to the introductory work to be done before writing a best-seller. This chapter focuses on the analysis phase of our sample application.

❑ **Chapter 4, “Designers at Work,” on page 81**

This chapter concentrates on the design phase.

❑ **Chapter 5, “Setting Up the Development Environment,” on page 105**

This chapter describes the preparatory work that paves the way for well-organized software construction. You are advised how to favorably initialize your new project in the WorkFrame environment.

❑ **Chapter 6, “Mapping Relational Tables Using Data Access Builder,” on page 131**

This chapter and the next two feature the Visual Builder! During the development of this book, we enjoyed most dealing with this tool and assume that you also will get excited when you read how we succeeded in implementing the sample application. You will reap the most benefit if you duplicate the implementation process step by step, following our instructions. In this chapter, you will use Data Access Builder to bring persistence to your application and enable your objects to be stored in a relational database.

❑ **Chapter 7, “Creating Visual Parts,” on page 147**

This chapter will guide you in developing the graphical user interface of your application, using the visual parts provided with VisualAge for C++. Most of the parts are used in our sample application, and you will be shown hints and tips to make the best of them.

❑ **Chapter 8, “Creating Nonvisual Parts,” on page 225**

Unlike other GUI development tools, Visual Builder allow you to develop your business object as nonvisual parts. In this chapter, we will show you how to develop the nonvisual parts that are used in the sample application.

❑ **Chapter 9, “Connecting the Parts,” on page 247**

Once you have built your visual and nonvisual parts, you are ready to draw graphically the connections between them. In this chapter, we show you how to connect the different parts to trigger messages from one object to another to let your application perform. Then, you just need to generate automatically the C++ source code of your application and compile it! Throughout Chapters 7 through 9, we focus on showing how to map your static and dynamic models from your detail phase to VisualAge for C++.

❑ **Chapter 10, “More about Visual Builder...,” on page 353**

If your curiosity is still not satisfied or if you want to take a closer look at some technical details, you should keep on reading. This chapter answers some questions that you did not ask before, such as: *What about the notification framework?* or *Can I reuse my legacy code?*

❑ **Chapter 11, “More about Data Access Builder...,” on page 373**

Accessing your database remotely from Windows 95, setting up DB2 Client Enabler, or using ODBC to transparently access text files as if you were accessing a relational database is what this chapter is about.

❑ **Chapter 12, “More about SOM...,” on page 421**

For those who want a comprehensive explanation of the SOM technology and its use through the Direct-to-SOM facility of VisualAge for C++, this chapter is made for you.

❑ **Chapter 13, “More about CDF...,” on page 453**

If you think that making your application OLE enabled is a lot more complex for you, wait to read this chapter. With few lines of code you can turn your application into an OLE container or an OLE server, thanks to the Compound Document Framework.

## Special Conventions in This Book

Several labeled boxes with predefined icons are used throughout the book to point out important information.



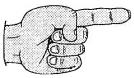
### ***Tips***

There are helpful tips scattered throughout the book. They contain information that can make your life easier.



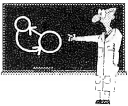
### ***Warning***

Always make sure to read the warning sections. They should draw your attention to areas where you can get trouble.



### ***Read This***

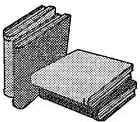
“Read This” icons point out things we did not want you to miss while reading over the chapters and developing your sample application.



### ***Technical Information***

Information located in these boxes complements the information we give throughout the chapters when building your sample application.

## Related Publications



The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

- ❑ *Object-Oriented Software Engineering. A Use Case Driven Approach* by I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. Addison-Wesley Publishing Company, 1992. ISBN 0-201-54435-0.
- ❑ *Object-Oriented Modeling and Design* by J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. Prentice Hall, 1991. ISBN 0-13-630054-5.
- ❑ *Designing Object-Oriented Software* by R. Wirfs-Brock, B. Wilkerson, and L. Wiener. Prentice Hall, 1990.
- ❑ *Modern Structured Analysis* by E. Yourdon. Yourdon Press, Englewood Cliffs, New Jersey, 1989.
- ❑ *Object-Oriented Analysis and Design with Applications* by G. Booch. The Benjamin/Cummings Publishing Company, 1994.

- ❑ *Object Technology in Application Development* by D. Tkach and R. Puttick. Benjamin/Cummings Publishing Company, 1994. ISBN 0-8053-2572-5.
- ❑ *Visual Modeling Technique—Object Technology Using Visual Programming* by D. Tkach, W. Fang, and A. So. Benjamin/Cummings Publishing Company, 1995. ISBN 0-8053-2574-3.
- ❑ *Object-Oriented Application Development with VisualAge for C++ for OS/2* by M. Carrel-Billiard, P. Jakab, I. Mauny and R. Vetter. Prentice Hall, 1996. ISBN 0-13-242447-9.
- ❑ *Effective C++: 50 Specific Ways to Improve Your Programs and Designs* by S. Meyers. Addison-Wesley, 1992.
- ❑ *OS/2 C++ Class Library, Power GUI Programming with C Set++* by K. Leong, W. Law, R. Love, H. Tsuji, and B. Olson. VNR Computer Library, 1993. ISBN 0-442-01795-2
- ❑ *Object-Oriented Programming Using SOM and DSOM* by C. Lau. VNR Computer Library, 1994. ISBN 0-442-01948-3
- ❑ *Open Database Connectivity (ODBC) SDK 2.10*, Microsoft Corporation, 1995.
- ❑ *DB2 Information and Concepts Guide*, IBM, 1995, S20H-4664.
- ❑ *DB2 Administration Guide*, IBM, 1995, S20H4580.
- ❑ *DB2 Application Programming Guide*, IBM, 1995, S20H-4643.
- ❑ *DB2 Glossary*, IBM, 1995.
- ❑ *VisualAge for C++ for Windows—Installation Guide and Product Overview*, IBM, 1996, S33H-5030-00.
- ❑ *VisualAge for C++ for Windows—User's Guide*, IBM, 1996, S33H-5031-00.
- ❑ *VisualAge for C++ for Windows—Programming Guide*, IBM, 1996, S33H-5032-00.
- ❑ *VisualAge for C++ for Windows—Open Class Library User's Guide*, IBM, 1996, S33H-5033-00.
- ❑ *VisualAge for C++ for Windows—Visual Builder User's Guide*, IBM, 1996, S33H-5034-00.
- ❑ *VisualAge for C++ for Windows—Visual Parts Reference*, IBM, 1996, S33H-5035-00.
- ❑ *VisualAge for C++ for Windows—Building VisualAge for C++ Parts for Fun and Profit*, IBM, 1996, S33H-5036-00.
- ❑ *VisualAge for C++ for Windows—Language Reference*, IBM, 1996, S33H-5037-00.
- ❑ *VisualAge for C++ for Windows—C Library Reference*, IBM, 1996, S33H-5038-00.



- ❑ *VisualAge for C++ for Windows—Open Class Library Reference Volume I, IBM, 1996, S33H-5039-00.*
- ❑ *VisualAge for C++ for Windows—Open Class Library Reference Volume II, IBM, 1996, S33H-5040-00.*
- ❑ *VisualAge for C++ for Windows—Open Class Library Reference Volume III, IBM, 1996, S33H-5041-00.*
- ❑ *VisualAge for C++ for Windows—Open Class Library Reference Volume IV, IBM, 1996, S33H-5042-00.*
- ❑ *VisualAge for C++ for Windows—SOM Programming Guide, IBM, 1996, S33H-5043-00.*
- ❑ *VisualAge for C++ for Windows—SOM Programming Reference, IBM, 1996, S33H-5044-00.*

## International Technical Support Organization Publications

- ❑ *Object Technology in Application Development, GG24-4290.*
- ❑ *Access to the DB2 Family with ODBC, GG24-4333.*
- ❑ *Client/Server Computing: The Design and Coding of a Business Application, GG24-3899.*

A complete list of International Technical Support Organization publications, known as redbooks, with a brief description of each, can be found as follows:

To obtain a catalog of ITSO redbooks, VNET users should type:

```
TOOLS SENDTO WTSCPOK TOOLS REDBOOKS GET REDBOOKS CATALOG
```

A listing of all redbooks, sorted by category, can also be found on MKTTOOLS as ITSOPUB LISTALLX. This package is updated monthly.

### How to Order ITSO Redbooks

IBM employees in the USA may order ITSO books and CD-ROMs by using PUBORDER. Customers in the USA may order by calling 1-800-879-2755 or by faxing 1-800-445-9269. Visa and Master Card are accepted. Outside the USA, customers should contact their local IBM office.

Customers may order hardcopy ITSO books individually or in customized sets, called GBOFs, which relate to specific functions of interest. IBM employees and customers may also order ITSO books in online format on CD-ROM collections, which contain redbooks on a variety of products.

## ITSO on the Internet



Internet users can find information about redbooks on the ITSO World Wide Web (WWW) home page. To access the ITSO Web pages, point your Web browser (such as WebExplorer™ from the OS/2 3.0 Warp BonusPak) to the following:

```
http://www.redbooks.ibm.com/redbooks
```

IBM internal users can download redbooks or read redbooks online. Point your web browser to the internal IBM Redbooks home page:

```
http://w3.itso.ibm.com/redbooks
```

If you do not have World Wide Web access, you can obtain the list of all current redbooks through the Internet by anonymous FTP to:

```
ftp.almaden.ibm.com
cd /redbooks
get itsopub.txt
```



The FTP server, *ftp.almaden.ibm.com*, also stores the sample from the accompanying CD ROM. To retrieve the sample files, issue the following commands from the */redbooks* directory:

```
cd GG244782
binary
get GG244782.ZIP
ascii
get READ.ME
```

All users of ITSO publications are encouraged to provide feedback to improve quality over time. Send questions about and feedback on redbooks to:

```
redbook@vnet.ibm.com or
REDBOOK at WTSCPOK or
USIB5FWN at IBMMAIL
```

To receive regular updates on new redbooks and general IBM announcements, you can subscribe to the IBM Announcement Listserver. It automatically supplies an Internet e-mail user with timely new announcement information (titles and optionally the letter or abstract) from selected categories. To get started, send an e-mail to:

```
announce@webster.ibm.link.ibm.com
```

The keyword “SUBSCRIBE” must be the only word in the body of the e-mail; leave the subject line blank. You will receive a category form and listserver details. To immediately start your subscription of, for

example, AS/400 announcements, put the words “SELECT HW120” in the body of the note. On the afternoon of an announcement day, you will receive e-mail with the announcement, along with a list of newly available redbook titles. To obtain a full abstract of a particular redbook, use “GET SG244782” in the note.

## VisualAge for C++ Support

VisualAge for C++ Service and Support is staffed by developers who handle everything from how-to's to complex technical problems. The resolution may take the form of education, a workaround, or a fix to the product (Corrective Service Diskette, CSDs).

There are several ways to contact the VisualAge for C++ Service and Support department electronically:

- ❑ **CompuServe™ forums:** GO VACPP
- ❑ **Internet**
  - anonymous logon to:  
*site:* [ftp.software.ibm.com](ftp://ftp.software.ibm.com)  
*directory:* [ps/products/visualagecpp/fixes/V35](ftp://ftp.software.ibm.com/ps/products/visualagecpp/fixes/V35)
  - sample URL:  
<ftp://ftp.software.ibm.com/ps/products/visualagecpp/fixes/v35>
  - Web site:  
[http://www.software.ibm.com/ad/visualage\\_c++](http://www.software.ibm.com/ad/visualage_c++)
- ❑ **Talklink (Windows Selected Fixes Area)**
  - 1-800-992-4777 for information (USA and Canada)
  - VACPP CFORUM
- ❑ **Developer's Connection (DEVCON) CD**
  - Ordering information: 1-800-6DE-VCON (USA and Canada)
- ❑ **IBM PC Co. BBS**
  - 1-919-517-0001 8,N,1
  - 1-800-772-2227 for information

## About the Authors

Marc Carrel-Billiard, from IBM France, works at the IBM International Technical Support Organization in San Jose, California. You can reach him by e-mail at [carrel@vnet.ibm.com](mailto:carrel@vnet.ibm.com).

Isabelle Mauny works in La Gaude (France) for the IBM EMEA Software Technical Support. You can reach her by e-mail at [ismauny@vnet.ibm.com](mailto:ismauny@vnet.ibm.com).

Michael Friess works in Stuttgart for the Developer Support Organization of IBM Germany. You can reach him by e-mail at [mfriess@de.ibm.com](mailto:mfriess@de.ibm.com).

## Acknowledgments



This book would not have been possible without the help of the following people who contributed information, resources, and technical advice: Ueli Wahli and Walter Fang, IBM ITSO San Jose; Jim Caldwell, Dimitry Feldshteyn, Rakesh Goenka, Maxine Houghton, Mike Polan, Barry Searle, Rollie Sing, and Brian Thompson, IBM Toronto; George DeCandio, IBM Research Triangle Park; Rainer Vetter IBM Germany.

Many thanks to Carmine Di Grande, ITSO San Jose Center Manager, Petter Sommerfelt, ITSO San Jose Center DM/ST Manager, and Barbara Isa, IBM Santa Teresa Lab, for getting this project started. Special thanks to everyone at the ITSO San Jose Center, in particular Elsa Barron, Mary Comianos, Stephanie Manning, Alan Tippet, and Guido De Simoni for their continuous support and to Andi Bitterer for always finding the right illustration at the right time. We are extremely grateful to Shirley Hentzell and Maggie Cuttler for their meticulous and unfailing review. Thanks also to Mike Meehan, Patti Guerrieri and Craig Little at Prentice Hall; and to Lou Evert at Software International Inc. for contributing to improve this book.

M.C.B.  
M.F.  
I.M.



# Part 1



## Introduction to the VisualAge for C++ Environment

*We are condemned to live in interesting times.*

-Chinese Proverb

Where are the good old days? Once, computer vendors provided not only mainframes with appropriate operating systems, but also matching software tools to extend the base equipment. Then, salesmen led happy lives supporting their two or three favorite customers. Application developers could focus on database transactions, concentrate their efforts on implementing the business logic, and forget about the user interface—because terminals behaved like typewriters, and the poor person who was allowed to give some piece of input felt like an external device, closely connected to the applications. Those days, when programmers as software gurus hacked thousands of lines of read-only code into their editors, are gone.

Look how times have changed! More people work out of their home offices, where they write at least a few lines of code in their preferred languages and fiddle with configuration files to tailor their individual environments. New technologies provide screens with brilliant graphical views of the user interface. Under constraints, programmers must build complex programs and be responsive to new requirements or changing environments. And most challenging of all, thousands of hardware and software suppliers freely offer their products but show little concern for connectivity.

Yes, those good old days have gone, and what we need right now are new tools and techniques to develop mission-critical applications that can run on various platforms and be easily adapted to new requirements. Otherwise, the software crisis will never end.

# 1

## **VisualAge for C++ and Application Development**

When we look at the manufacturing industry, we find that many manufacturers use components to build their products. We discover that many standardized elements, such as bolts and nuts, can be purchased anywhere. We learn that companies use the same component for different products; for example, car manufacturers use the same rear-view mirror for all of their models or the same clutch for many of their models. We realize that, before going into actual production, engineers build a mock-up that reveals possible construction faults.

When we look at the software industry, we find that many new products are built from scratch, and for a number of reasons: A new programming language appears that is supposed to easily solve problems of a certain domain, an application requires a new database system that causes many changes in existing software, a new team member arrives with new and better ideas, or an old team member leaves the company, accompanied by all of his or her undocumented knowledge.



We discover that there are no standardized software modules on which programmers can rely. We learn that there are many function collections, so-called program libraries, that help deal with various software domains, such as databases, networks, communications, and graphical user interfaces (GUIs). If programmers want to use those libraries, however, they must laboriously look for each function and its parameters, leafing through multivolume manuals. Furthermore, if programmers mix libraries from different producers, they are often confronted with compatibility problems, such as duplicate names.

VisualAge for C++ does not do away with low-level function libraries and cannot prevent library producers from using the same names, but it supports the building of well-designed models and software parts that can be reused in multiple applications and on different hardware and software platforms.

## Visual Programming

During the past 10 years, software designers have enriched the presentation of operating systems on personal computers and workstations, providing users with GUIs. At the same time, software developers have begun to accommodate their applications to this new environment.

The benefits of GUIs from the user's perspective are obvious:

- ☐ Users no longer have to type command lines with many arguments and cryptic options.
- ☐ Users can control applications more intuitively.
- ☐ Users can simultaneously look at different views.
- ☐ Applications look polished and provide a consistent interface.

Programmers, however, must deal with hundreds of new functions that exploit the capabilities of the GUI, and they must cope with a new programming approach: event-driven programming.

In the event-driven programming paradigm, programmers send messages to graphical elements, and, if an event occurs, the graphical system sends a message to a function that programmers must provide. So, from the developer's perspective, the disadvantages of a GUI also are obvious:

- ☐ New concepts must be learned quickly.
- ☐ Complexity increases.
- ☐ Thus, development time increases.

To shorten both the learning curve and development time, some large and small software companies alike offer tools that enable programmers to develop applications visually. Thus, programmers do not have to invoke their editors to start writing a new program; they can build the GUI by designing it on their screens. But, of course, there is more than the GUI—programmers must be able to add business logic and data access transactions. The trouble begins exactly at this point. With most existing tools, programmers can no longer develop visually; they must provide the code manually. Using VisualAge for C++, however, programmers can continue to work visually, because they have the components for building not only a GUI but also the entire application, including database access and multimedia features.

Before you can enjoy the powerful tools of VisualAge for C++, you should be acquainted with object-oriented application development, an approach that has begun to emerge as the software world becomes more and more complex. The GUI challenges us, as does the need for remote data access, with its underlying communication protocols, and the fact that we cannot quickly rewrite existing code to adapt it to a new hardware or software environment. We need methods and tools to help us comprehend and deal with this challenging complexity.

Instead of decomposing huge applications into procedures, today's software specialists understand problems as assemblies of objects. This approach simplifies their views of problems and helps translate those views into software. Object-oriented languages, in which the concept of objects is inherent, support programmers in their translation efforts.

Programmers do not necessarily have to use an object-oriented language; they could implement objects and their behavior by using a procedural language. Procedural languages, however, involve a certain degree of danger; namely, they do not hinder programmers from arbitrarily accessing objects. Programmers can directly modify an object's data during the execution of every module, so they must alter many modules whenever an object changes its behavior or data structure.

With object-oriented languages, programmers can access an object's data only within certain modules, so they know where to apply the changes. In addition, because the object-oriented approach includes an analysis and design phase that programmers must go through before they start writing programs, they are unlikely to write poorly structured code. In this book we explain all phases of the object-oriented approach and show you how to put the approach into practice with the help of VisualAge for C++.

## Object Talk

In this section, we introduce some object-oriented terms and concepts that we use throughout this book. You will not find an in-depth discussion of the object-oriented approach to software development or the definition of every object-oriented term. If you are interested in an extensive explanation of object-orientation, we recommend that you consult the books listed in the Related Publications section.

### Objects

In our real world, an object is, according to *Webster's Dictionary*:

*Something perceptible, especially to the sense of touch or vision.*

Indeed, according to this definition, we have a large assortment of objects! Let us take a car as an example. If you ask two people to describe a certain car, they will probably give two completely different answers, on the basis of their knowledge, their way of looking at and evaluating things, and their interests. A passionate driver will tell you about the car's motor and give you many other technical details about the internal workings of the car. A person who has never driven a car will tell you about the color of the car, its estimated length, width, and height, and anything else that is visible. No one, however, can give an exact description that correctly encompasses all properties of a car. Even a car's manufacturer, who knows every element of the car, would fail to describe it correctly, because he or she would not know its current mileage or the amount of missing tire rubber. Objects in our real world have an infinite number of attributes and purposes. A car's purpose is as a driving machine, but it also can be (mis)used, as a dog house, for example.

In driving schools, instructors describe a car by emphasizing its function and explaining how to handle the steering wheel, gearshift, and pedals and how to interpret the indicators on the dashboard. In technical terms, we can say that instructors explain the interface of the car, how different input parameters influence the car's behavior, and how drivers can understand the car's output values.

So we see that real-life objects have various properties, namely:

- ☐ Attributes, such as color, length, width, height, weight, and mass
- ☐ Interfaces, such as steering wheels, pedals, and door handles
- ☐ Functions or actions, such as driving, braking, or sheltering dogs

By now we should have some idea of how we can realize the transition from our real world to the virtual world of computers. Of course, the object-oriented approach did not invent this transition, but it facili-

tates it in a neat and transparent manner. Procedural programming languages offer primitive data types, such as integers, floating-point numbers, characters, and pointers. Additionally, they offer compound data types, so-called structures or records, where several elements with primitive or compound data types can be stored. Object-oriented languages offer a particular data type that can store all properties of an object and guarantee that the object's attributes are manipulated only by its own functions or well-defined interfaces.

In the procedural approach, an object generally is divided into structures that contain its attributes and procedures that deliver its functions and operate with its data. The disadvantage of using procedural languages is that nothing (except rigid discipline) prevents programmers from directly manipulating the data of other objects. Although such manipulation was not originally intended, programmers are inclined to do so when there are time constraints. Therefore, the procedures and data structures of different objects often are heavily interdependent, which makes reusability almost impossible and complicates the process of extending the software system.

Objects in computer environments, once they are designed, are discrete, have a limited number of attributes, and behave as defined (but sometimes not as intended). Software objects represent real-life objects but are implemented in a manner that is appropriate for a particular problem. Again, let us take a car as an example of an object. For an application that supports car sellers, designers implement the car object with attributes that are important for marketing purposes, such as price, horsepower, color, and number of air bags. These attributes are rather high level, because customers usually are not interested in the amount of steel or aluminum that was used to produce the car. For an application that supports the car manufacturing process, however, designers must assign more attributes to the car object, because engineers are interested in details that are essential to the building of a car. In both applications, the car object adopts only some of its real-world attributes.

## Classes

If we consult Webster's dictionary again, we find that a class is defined as:

*A set or group whose members share at least one attribute.*

This definition also offers unlimited possibilities. Let us take, for example, animals, as all members share the haveOrganism attribute. Biology teaches us that humans also share the haveOrganism attribute, so according to Webster's definition, humans belong to the

same class. Sorry? Another example: car is a class, as all members have the fourWheels attribute. Rollerskates also share the fourWheels attribute, so they belong to the same class. Sorry again?

These examples show how difficult it sometimes is to find the correct level of abstraction. Here, of course, it is quite obvious that we chose the wrong level. Animals and humans do belong to the same class, but to a more generalized class, say, livingBeings. Rollerskates and cars can belong to the more generalized class, fourWheeled. We notice that we can describe classes in more abstract terms than we can describe objects.

We chose our first example of a class from science, not accidentally, as biologists actually group the world of living beings into classes, such as mammals, cold-blooded animals, and microbes. Chemists group substances into classes, for example, organic and inorganic substances, and physicists deal with solid, liquid, and gaseous substances. We see that in every domain scientists use classes to structure our complex world into comprehensive groups. Depending on the current focus of our interest, the classification may be very detailed.

Classes in computer environments are invented to structure a software problem and thus should be regarded as groups whose members share several (not just one) attributes or provide similar function. In our examples of both objects and classes, we look at cars. The Car class contains the set of all cars. Every car provides the same function and has similar attributes and interfaces. This fact is quite obvious because experienced drivers can operate every car entrusted to them, even if they have never seen a certain model before. The Car class describes the general functions that all cars have in common. A car object is a distinct instance of the Car class, such as your car or the car parked in your preferred space.

## Inheritance

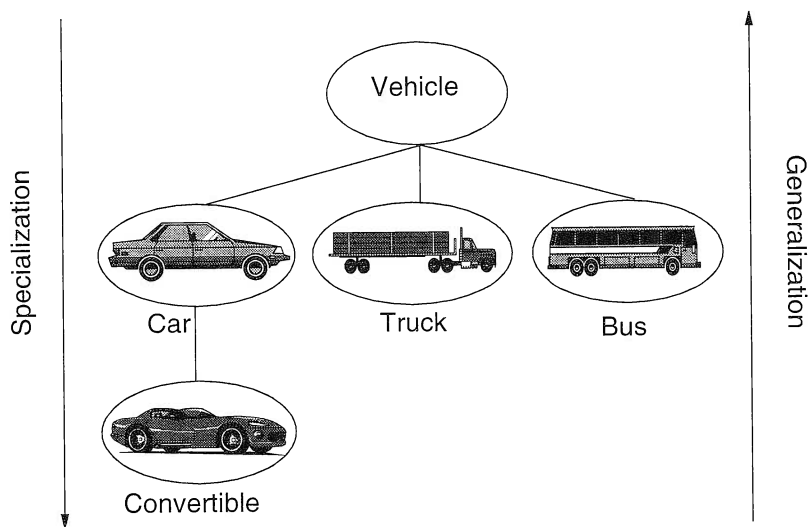
In real life we know the term “inheritance” very well, as we all dream of coming into a small fortune through inheritance. Its meaning in the object-oriented paradigm is completely different, however.

Strictly speaking, inheritance in the object-oriented world relates more to its biological meaning, because one class does not bequeath its properties to another class and pass away; the new class originally looks and behaves like its parent class, and the two classes coexist. The class designer specializes the new class by adding or changing the original attributes or functions, so that the class can fulfill its intended purpose. You can compare this work to that of a car design engineer who wants to create a new model. He or she probably does

not begin from scratch but takes an existing car as a prototype, modifies it here and there, perhaps removing the roof to come up with a convertible version.

Note that the design engineer designs the new model by using drawings, or better, with the help of a computer-aided design (CAD) program. Consider also that the removal of the roof compels many subsequent changes. We can assume that a car without a roof requires a different chassis, different front doors, and other parts that will differentiate it from a car with a roof. So, we do not recommend applying inheritance when the derived class undergoes this kind of change, because the main benefit of inheritance (and here we come back to computer science) should be reuse of existing classes.

Look at Figure 1. The Convertible class is a specialization of the Car class, and the Car class is a generalization of the Convertible class. A generalization of the Car class would be Vehicle. Inheritance means that the specialized class adopts all properties of its ancestor class; that is, the specialized class behaves exactly like the ancestor class and owns the same attributes. In fact, if you derive from an existing class, you work on its copy. But, you can add new attributes and functions to the derived (specialized) class or change existing functions. For example, the `openSunRoof` function is no longer valid for the Convertible class, whereas the `rollbar` attribute is not present in the Car class.



**Figure 1.** Inheritance

The ability of the descendants of a class to inherit all functions of their ancestor provides the means of reusing code. Modification of common behavior need be implemented only once, namely, inside the functions of the ancestor class.

## Encapsulation

The implementation details of a car's functions are hidden from drivers, who must know only how to handle the interfaces.

Car drivers should treat their engines with care and shift the gears appropriately. For beginners, the procedure of depressing the clutch, shifting gears, letting out the clutch, and letting in the clutch again is most challenging during the first few lessons at a driving school. In cars equipped with automatic transmissions, drivers can shift gears without having to use any additional pedals or shifts. The gear change is encapsulated inside the acceleration process, so that drivers need only operate the gas pedal. Drivers also know that as soon as they apply the brakes, the car slows down, but they do not know (and do not care) whether the brake pedal activates a disk brake or a drum brake.

Object-oriented programming languages as well offer encapsulation. Objects reveal only their interfaces, not their internal implementation. Callers of the functions do not care how the underlying algorithms are implemented, but they rely on the promised behavior of the function.

## Polymorphism

The term *polymorphism* looks so strange that we consult Webster's again and find it defined as:

*Genetic variation that produces differing characteristics in individuals of the same population or species.*

Well, this definition in itself looks strange. So let us try to understand the meaning of polymorphism by looking at its Greek root and constructing a noun phrase from the result: *poly* means many or multiple, *morph* means shape or form. So, we define polymorphism as:

*Ability of a thing or organism to exist in multiple forms.*

In the real world, we can find such examples as wax or amoebas. When you visit wax museums you can see that wax really exists in many different forms. And if you happen to have access to a microscope and a culture of microorganisms, you can see that amoebas constantly change shape as they move and engulf food.

Polymorphism in an object-oriented sense differs slightly from wax and amoebas. The characteristic of having different forms can also be interpreted as providing flexible, that is, nondetermined, behavior. Let us take a crisp example from real life: When you step on the tail of a dog, it barks, whereas when you step on the tail of a cat, it meows. With all apologies to the pet world, this is polymorphism in action! The same action executed on different species (understand type of object) provokes different reactions. Let us look at how polymorphism applies in your everyday life as a programmer.

Generally, when you invoke a function, you expect a determined flow of execution, because the function is designed and implemented to carry out a particular task. In pure object-oriented languages, however, a function is always coupled to a class. At coding time, the exact class that is coupled to the function when it executes need not (and often cannot) be known. Then, during run time, when the function is called, the class that actually executes the invoked function can be the specified class or a descendant of that class.

Say, for example, that we want to call the draw function of the Figure class, and we expect that the actual object should draw itself on the screen. We do not care, however, whether during run time the actual object will be an instance of the Circle or Square class, both of which are descendants of the Figure class. Obviously, the draw function behaves differently according to the actual class. In future releases of the application, one or more new descendants of figure might exist, for example, the Triangle class, which also provides its draw function. The good thing is that the caller of the draw function does not have to know about the existence of the new class.

Developers cope with many classes and objects. Some classes and objects directly represent the image of real-life objects; others are metaphors for services that are required to implement the business logic or communicate with either the user of the application or external devices. In large software applications, the associations and interactions among all objects are both difficult to describe and complex, so we must have methods to shed light on that complexity. Without such methods, developers would soon see themselves as “object-disoriented”!

## Object-Oriented Methods

Webster's defines the term *method* as:

*Orderly or systematic arrangement, sequence, or the like.*

In fact, we need a systematically arranged model to define, refine, implement, maintain, and document complex software constructions. It is important that all participants of a project know the terminology



of the problem domain. Generally, when you begin a new software project, you are given some ambiguous text or informal specifications. Your customers cannot express precisely what they want, and, if you do not know everything about their specific problem domain, you cannot ask the correct questions to fill in the gaps. As soon as development starts, the requirements for the product change, because some gaps now become obvious, and you can hardly estimate how long you will work on the implementation. In most cases, development goes on indefinitely, because users always find something that is worth changing or adding. Object-oriented methods cannot prevent your customer from having additional requirements, but it can decrease the effort you expend to integrate the extensions into your design and implementation.

Several analysts, such as Rumbaugh, Jacobson, and Wirfs-Brock, have published techniques for translating real-life problems into different models that offer a view of the problem domain and facilitate system implementation. Because object-orientation is a rather new subject, some of the methods are likely to be refined in future publications. One common thread among the methods that is not likely to change, however, is the recommendation to develop applications iteratively. The visual modeling technique (VMT), which we introduce next, has adopted the object-oriented methods of Rumbaugh, Wirfs-Brock, and Jacobson.

James Rumbaugh's object modeling technique (OMT) is popular because of its simple notation. Basically, OMT consists of three models: the static model, which captures the relations among objects; the dynamic model, which captures the run-time behavior of objects; and the functional model, which sketches the flow of operations.

Rebecca Wirfs-Brock's responsibility-driven design (RDD) reflects the responsibilities, that is, the tasks, that a class must accomplish. Ms. Wirfs-Brock introduces collaborators, which are classes that help a class fulfill its responsibility. She suggests creating one class-responsibility-collaborator (CRC) card per class; each card indicates the class, lists all of its responsibilities, and for each responsibility gives the related collaborators.

In Ivar Jacobson's object-oriented software engineering (OOSE) technique, objects and classes are found with the help of use cases. A use case is an external view describing an interaction between a user and a system. The technique essentially draws a border around the problem domain and defines user roles.

Rumbaugh, Jacobson, and Wirfs-Brock state that there are static and dynamic models. The static model, also known as object models, focuses on the hierarchy and associations of objects. The dynamic model emphasizes the interdependencies and run-time behavior of objects.

# Visual Modeling Technique

In Webster's we find the term *technique* defined as:

*The systematic procedure by which a complex or scientific task is accomplished.*

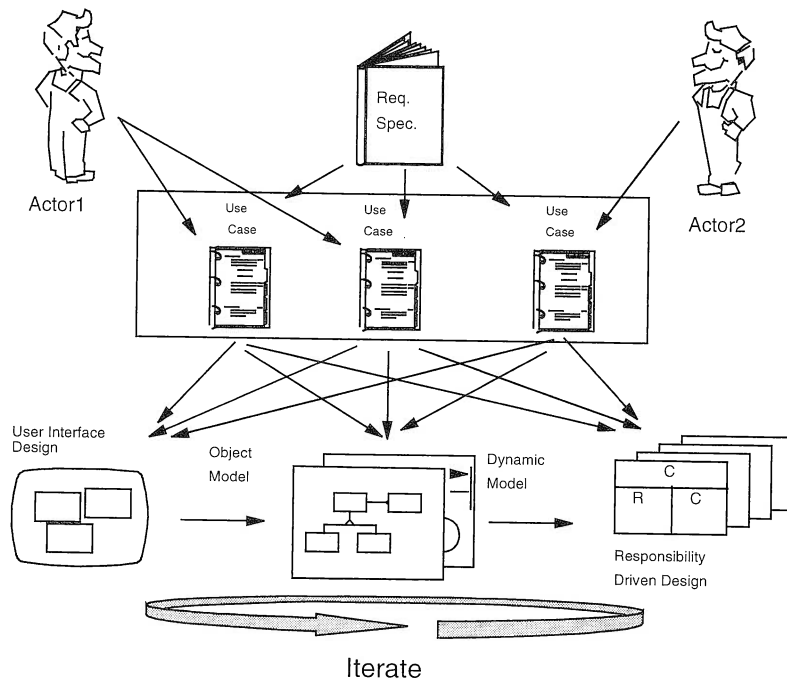
Software development is actually a complex task. Although several attempts have been made to approach the task scientifically, with the goal of automating software development, the inherent complexity is (still) a big obstacle. We need heuristic and iterative techniques to master the problem.

If we look at the methodologies of Rumbaugh, Wirfs-Brock, and Jacobson, we notice that they thoroughly explain their respective systematic procedures, but none of them address the new programming technique that has become important for modern application development, namely, visual programming. The authors mention the importance of developing an analysis prototype that customers can use to verify the correctness and completeness of the user interfaces, and they demand the development of a design prototype that reflects the current state of the design model and evolves toward the final implementation. But, they do not explain how to implement the prototypes.

VMT fills the prototype implementation gap and, by using the following methods or techniques, serves as a roadmap for developing applications with GUIs:

- OOSE: VMT uses the use case model to find potential objects and classes in the problem domain that form the object model. The use case model also serves as a starting point for the development of dynamic models.
- OMT: VMT uses the static model (object model) and the dynamic model (event-trace diagram and state-transition diagram) to illustrate the relationships among objects and the run-time behavior of the objects.
- RDD: VMT uses the CRC card technique to identify an object's responsibilities and collaborators.

VMT is a complementary approach to object-oriented application development (Figure 2). It uses OMT notation to illustrate the static and dynamic models, and it divides the development process into three phases: analysis, design, and implementation.



**Figure 2.** VMT: A Complementary Approach to Object-Orientation

Furthermore, VMT proposes that you not draw strict boundaries between the phases; rather, you should analyze a little, design a little, implement a little, verify your results, and iterate. With VMT, you can check whether the resulting models are complete and consistent and thus provide a stable and verified system.

## Analysis

The goal of analysis is to understand the problem domain, that is, to clarify *what* the system should provide. So, the first step is to separate the problem domain from the real world. In most cases, you would carry out this first step together with your users and define problem statements that are based on the requirements specification. Then you would arrange these problem statements to form use cases (*Object-Oriented Software Engineering. A Use Case Driven Approach*, by I. Jacobson et al.). At this stage, the use cases are rather high level. You do not consider any implementation constraints, except that the system should be affordable and implemented within a reasonable time frame. You simply describe the essentials of the system's functions, regarding the functions as black boxes. Consequently, you take only

those objects that directly represent their real-life counterparts; you can find these objects by analyzing the use cases (the process of finding objects is discussed in Chapter 3, “Analysts at Work,” on page 59).

You develop an analysis prototype and show it to your users, so that they can verify that the use cases are complete and correct. If the use cases are not complete and correct, seize the opportunity: extend existing use cases or add new problem statements and formulate new use cases that were not obvious at the very beginning. Then, refine the prototype and go back to your users.

When you have finished developing the use cases and the corresponding user interfaces, determine whether you can group some of the objects in classes. Once you have found all classes, you can establish their relationships (especially inheritance and aggregation), attributes, and behaviors. The dynamic model describes how the objects interact. As the objects of the analysis model are derived directly from the problem statements and therefore represent real-life objects, they are also called *semantic objects*.

We discuss the analysis phase in detail in Chapter 3, “Analysts at Work,” on page 59.

## Design

The main goal of the design phase is to devise a solution, that is, answer the *how* question. As input you use the models that you developed during the analysis phase. VMT divides design into system design and object design. In system design, you determine the hardware and software components that are relevant for the application, such as the operating system, programming language, development tools, database system, and communication protocol. You also chart a high-level structure for the application functions. In object design, you refine the models from analysis, considering the constraints that the hardware and software components impose on the system. You then use these refined models for the design prototype.

We explain the design phase in detail in Chapter 4, “Designers at Work,” on page 81.

## Implementation

The goal of the implementation phase is to translate the design model into the implementation model, that is, the actual application construction. The design and implementation phases are closely coupled as the design prototype gradually evolves toward the final implementation model.

We explain the implementation phase in detail in Part 3, “Building the Visual Realty Application” on page 103.

## Visual Programming with VisualAge for C++

VisualAge for C++ takes advantage of the visual programming construction technology of IBM’s VisualAge Smalltalk™ product. VisualAge for C++, a follow-on product of the former CSet++™ product, includes the development tools from the CSet++ product. This powerful, object-oriented combination lets you build parts visually and then combine the parts to construct sophisticated applications. The key concept of VisualAge for C++ is that all existing or built parts are designed for reusability.

Using VisualAge for C++, even inexperienced programmers can build partial or complete applications because they do not have to write any code, provided that all components already exist. These components consist of graphical elements, the so-called visual parts, and the classes that handle business logic and data access, the so-called nonvisual parts.

In truth, only in rare cases will you develop an application without having to add a piece of code! If you must add some code manually, however, VisualAge for C++ supports you in adhering to the object-oriented paradigm, that is, building reusable parts. It provides a library of prefabricated, ready-to-use components that you can use “as is” or enhance. You decide whether you write the missing code yourself or buy additional libraries with parts that meet your needs.

Once you start using VisualAge for C++, you gradually learn the concepts of object-orientation. At first, you might take advantage of the GUI creation capability only, while still calling your existing code. Then, you take the plunge into visual programming, reusing your GUI. Finally, having improved your skills in object-orientation and C++, you explore the advanced features of the product that enable you to create your own parts.

When you program visually with VisualAge for C++ you develop your applications by using a graphical, not a textual, tool. The Visual Builder tool provides a powerful framework for developing not only simple GUIs but a complete application. You choose the visual parts that your application requires and, using the mouse, lay out the application interface by dragging and dropping the parts on a free-form surface. In this way, you and the end user of the application can look at the GUI before you have included the business logic and invoked the compiler.

Before we show you how to build an entire application by using an object-oriented approach, let us describe the complete set of VisualAge for C++ tools.



# 2

## **Getting Started in a VisualAge for C++ Environment**

We recommend that you read this overview of the VisualAge for C++ environment if you want to develop a broad understanding of the VisualAge for C++ tools. In this chapter, we also introduce the basic concepts of the tools that are used to develop the Visual Realty application, such as the Visual Builder or Data Access Builder.

From this point and throughout the following chapters, we use the term VisualAge for C++ to address VisualAge for C++ for Windows version 3.5.



## A Short Overview

VisualAge for C++ is a member of the VisualAge application development family. The Windows 95 and Windows NT platforms are both supported as development and run-time platforms. You may also use the Windows 3.1 platform as a run-time platform if you have installed the Win32s interface (version 1.3). VisualAge for C++ comes with the Windows Software Development Kit (SDK), which contains various libraries and header files, as well as some development tools such as an OLE2 object viewer or a Dynamic Data Exchange (DDE) spy. VisualAge for C++ also comes with REXX™ for Windows™.

VisualAge for C++ provides you with all of the tools you need, from doing rapid application development (RAD) with Visual Builder to tuning application performance with the Performance Analyzer. VisualAge for C++ lets you manage your project, build your application with the Visual Builder and Data Access Builder, edit the code, compile it, and finally link it. The Debugger eases the task of fixing problems. The Browser and Performance Analyzer, respectively, help you understand the structure of your code and the behavior of your application at run time. Of course, if you do not want to use Visual Builder to build your application, you can still type in your source code and take advantage of the powerful VisualAge for C++ compiler and linker.

Within VisualAge for C++, tools can interact through the WorkFrame services. An environment with this capability is called an *integrated development environment* (IDE).

## Managing Your Project

Good programming discipline suggests that you not start coding too quickly—as soon as you have a rough idea of what your application should do, for example. Rather, you should first plan your development and organize files on your system. The WorkFrame component can help you with these planning and organization tasks.

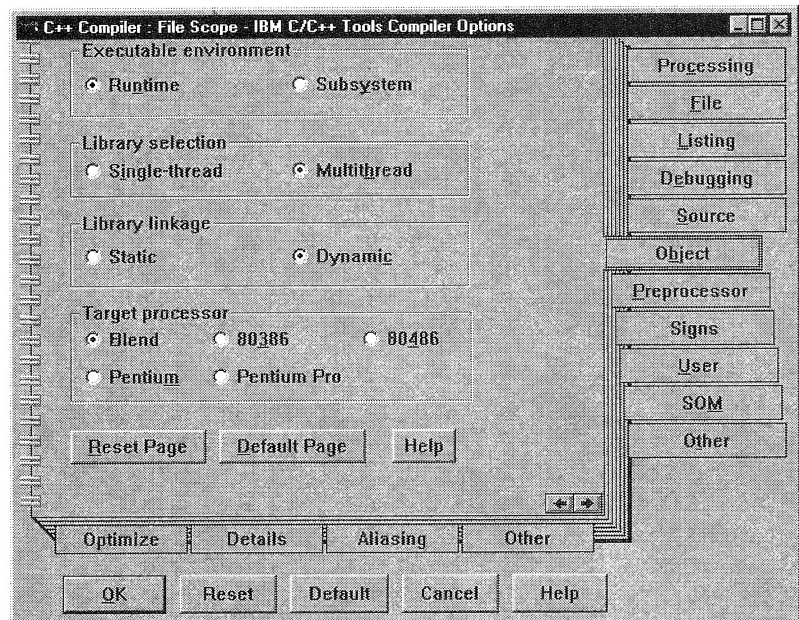
### WorkFrame Concepts



When building an application, you deal with many different pieces of data, such as C++ source files, resource files, and help files—the project elements. All project elements that make up an application or a subsystem are grouped inside a project, which is the core of the WorkFrame environment. Each project has a single target, for example, an executable file or a library.

Each project element has a type. WorkFrame uses this type to choose which action to apply to a project part. For example, if you define a C++ SourceFile type as all files matching the \*.cpp file mask and a SystemEditor action that takes the C++SourceFile type as input and corresponds to the LPEX Editor, all project elements with the .cpp extension can be edited through the LPEX Editor.

An action can correspond to any file that can be run, such as an executable file or a command file. An entry point is associated with each action and located into a specific support dynamic link library (DLL). This DLL defines the action default options and provides the GUI to easily alter them. For example, all VisualAge for C++ compiler options can be accessed through the interface shown in Figure 3.



**Figure 3.** Interface to Alter Options for the VisualAge for C++ Compiler

An action can be file or project scoped. File-scoped actions can be invoked only from the project elements defined as their source types. For example, the IconEditor action is available only from icon files. Project-scoped actions apply to the project entity and can generally be invoked from any tool that has been started within the project. A typical example of a project-scoped action is the Make action.

## Creating a Project with WorkFrame

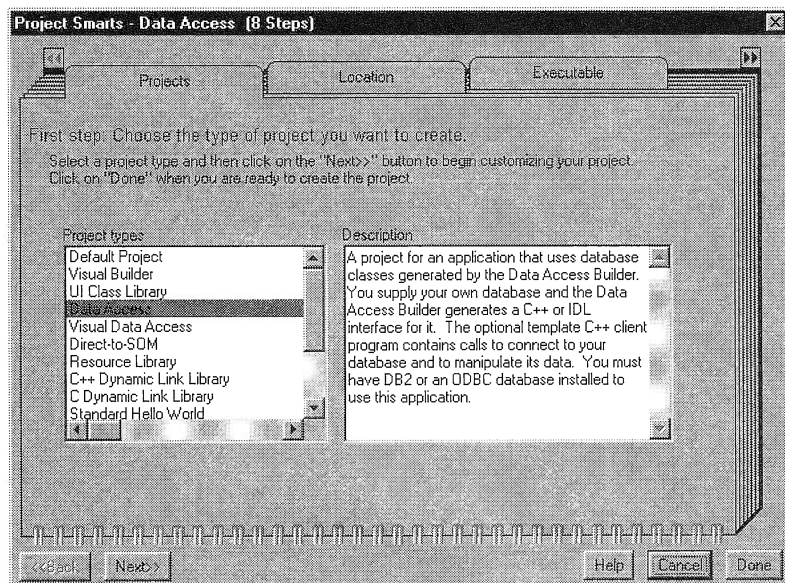
To take full advantage of the integration facilities of WorkFrame, you must create a project for your application or subsystem, either by copying an existing project and modifying its settings, or by using the Project Smarts facility.

### *Using Project Smarts*

Project Smarts is a facility that offers a catalog of skeletal applications to use as a quick start for your project. With Project Smarts, you can create a project configured according to an application category, such as:

- ☐ Visual Builder application
- ☐ Presentation Manager application
- ☐ Resource Dynamic Link library
- ☐ C++ Dynamic Link library
- ☐ Data Access Builder application

Choosing a certain catalog entry starts a notebook dialog that lets you specify the project settings, such as its working directory and name, and then creates a project configured with the compiling and linking options appropriate to the application. Each skeletal application consists of some project parts that you can use as a basis for your own application development. Figure 4 shows the description of the Data Access skeletal application.

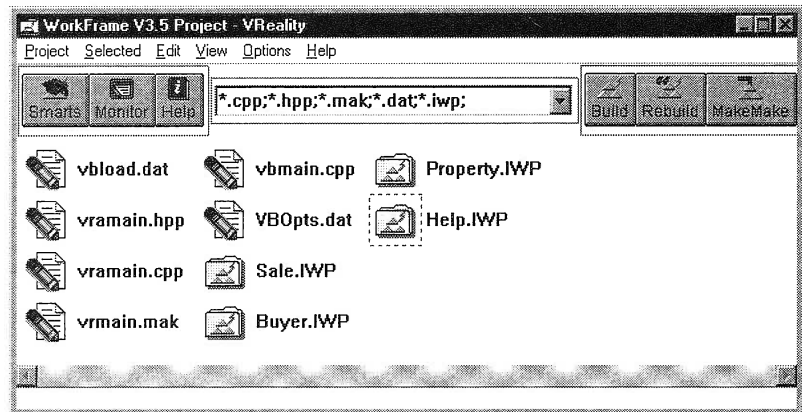


**Figure 4.** Project Smarts Catalog View: Data Access Application

Any application created from Project Smarts inherits from the VisualAge for C++ default project actions and types. Those actions and types are defined in the file `VACPP.IWS` located in the `D:\IBMCPW\MAINPRJ` directory. In “Modifying the WorkFrame Configuration File” on page 123, we show you how to modify this file manually and add new types and actions to the default project.

## Creating Composite Projects

Most applications, unless they are truly simple, consist of a hierarchy of projects. The way in which you organize your projects reflects their dependencies. Defining a project as composite is equivalent to creating other projects as project parts of that project. Typically, an application can be divided into several subsystems. For each subsystem there is a corresponding library, which you must build before building the application itself. You could build such an application, as depicted in Figure 5, where the main project depends on its nested projects; that is, you can specify that you want to build the nested projects prior to building the main project.



**Figure 5.** Example of Composite Project

WorkFrame handles composite projects in such a way that the Build and MakeMake facilities recursively build the project hierarchy.

## The MakeMake and Build Facilities

Used together, the MakeMake and Build facilities let you build your application without having to create and maintain make files. The MakeMake facility creates a make file for a project by examining the tools setup actions and types and determining the correct sequence of commands to build the project target.

The Build facility uses the MakeMake facility to build the make file for your application and to start a make utility such as nmake against the generated make file. The Build facility understands project organization and thus builds subprojects first in a project hierarchy

## Customizing a Project with Build Smarts

With the Build Smarts facility, you can temporarily modify the requested compiling flags and linking options for the most common build options, such as debug, browse, or optimize. Build Smarts overrides the current options for the compile and link actions as defined in the project.

When you are working with composite projects, Build Smarts lets you specify whether you want to build subprojects first. This specification prevents the Build facility from trying to recursively build all projects in the project hierarchy. You can also specify preprocessor macro values to be defined or undefined at the development or production stages.

## Generating Your Code

In this section we introduce Visual Builder, a tool for visual programming; the IBM Open Class Library, which provides building blocks for your application; and the Data Access Builder, which maps relational database tables to C++ classes.

### Using Visual Builder



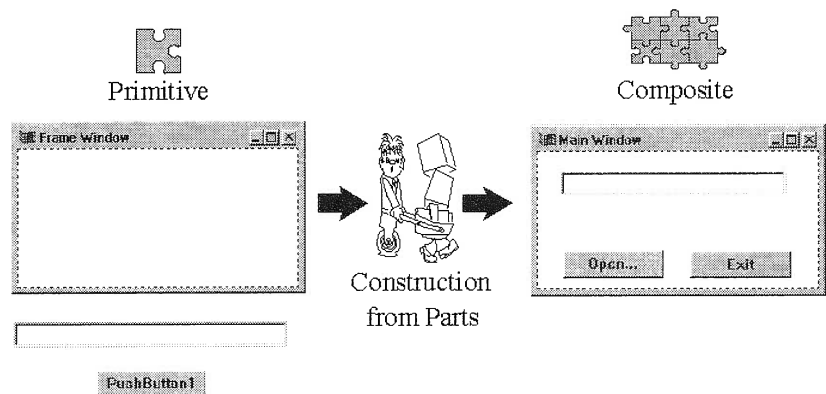
Traditional GUI builders let you create the interface of your application and generate the code for that interface. They do not provide a visual way of generating the behavior of your application, such as the piece of code executed when you click on a push button.

Visual Builder is not a traditional GUI builder. It lets you create a complete application visually by reusing parts, connecting them, and generating the code for the entire application. The generated code uses the IBM Open Class Library and therefore is portable across the platforms where the library is available. See “Building from Blocks” on page 35 for more information about the IBM Open Class Library.

## Visual Builder Concepts

Just as you would use building blocks to build a wall, Visual Builder uses parts to build applications. You can think about parts as reusable components that you can tailor to fit your needs, just as you would cut a building block to fill a gap in your wall.

Any application made from parts is a part itself: Assembling primitive parts results in a composite part. A primitive part can be a window or an entry field; a composite part can be a complete panel for a database information update (Figure 6). The composite part can be reused as a building block in another application.



**Figure 6.** Primitive and Composite Parts

Parts are either visual or nonvisual; an entry field and a frame window are examples of visual parts; a list of customers is an example of a nonvisual part.

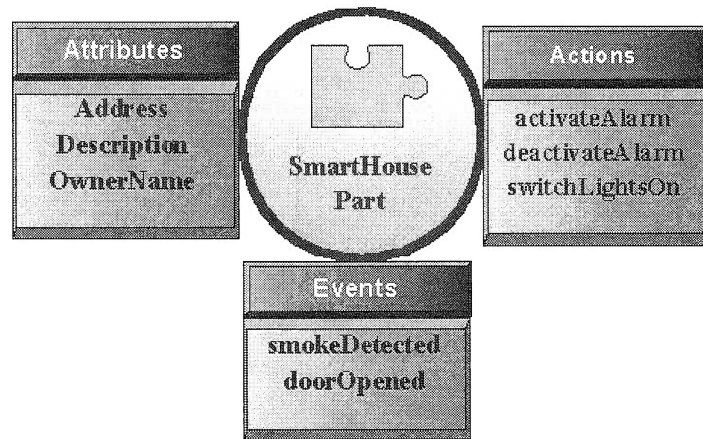
**Parts Interface.** Parts communicate through their interface. A part interface consists of three features: *attributes*, *actions*, and *events*. These features correspond to a natural way of viewing parts in terms of the properties they have (attributes), the services they can provide (actions), and the notifications they can send (events). Figure 7 shows a sample part interface for a nonvisual part called SmartHouse.

### Technical Information



A part is a C++ class. However, it has properties that a conventional C++ class does not have, such as notification enabling. If you map a class to a part, the data members of a class correspond to the attributes of a part and the methods of a class correspond to the actions of a part. An event is a particular feature of a part that triggers a notification.

The SmartHouse nonvisual part manages a so-called intelligent house that can detect when someone enters the house and can monitor a smoke detection system. SmartHouse has been designed to send events if it detects anything unusual (doorOpened and smokeDetected events) and start actions, such as activating the alarm or automatically switching on the lights.



**Figure 7.** Sample Part Interface: SmartHouse

**Connecting Parts.** Connections define how parts interact through their interface. A connection is a one-to-one visual relationship between two parts (visual or nonvisual). Connections are categorized as:

#### **Attribute-to-attribute**

Whenever the value of the first attribute is changed, the value of the second attribute is updated, so the attribute values are always the same.

#### **Event-to-attribute**

Whenever an event occurs, the attribute is updated.

#### **Event-to-action**

Whenever an event occurs, the action is performed. A variation of this, the *attribute-to-action* connection, starts an action when a certain attribute event (for example, attribute changes value) occurs.

## Event-to-custom logic

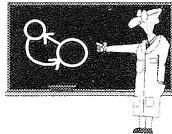
An event-to-custom logic connection lets you call some user code when the event occurs. The customized code is encapsulated in a `codeSnippet()` function. A variation of this connection is the *attribute-to-custom logic* connection.

## Event-to-member

Whenever an event occurs, a member function of the currently edited part is called. This connection lets you call any member function, even if it has not been added to the part interface. A variation of this connection is the *attribute-to-member* connection.

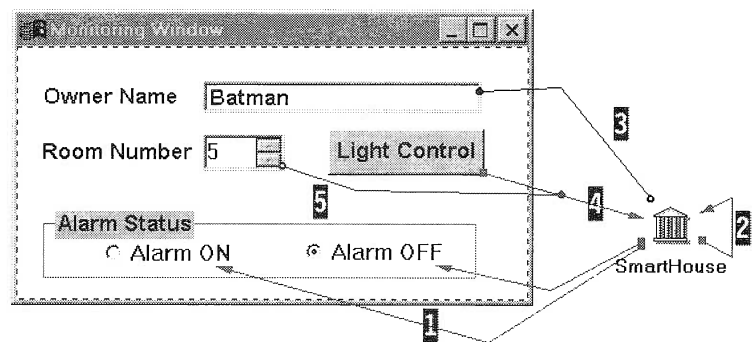
In Visual Builder, the origin of a connection is called the *source* part and the destination of the connection is called the *target* part.

### Technical Information



Whenever you use an attribute-to-attribute connection, the way in which you draw this connection between two parts is not without consequences: the direction of the arrow determines which is the source and which is the target at *initialization* time. That is which attribute value is used as a reference to set the value of both attributes.

Figure 8 shows several connections for the SmartHouse nonvisual part. In this example, we create a simple GUI to monitor the state of a SmartHouse.



**Figure 8.** Sample Connections: SmartHouse Monitoring Window



We first build an entry field (Owner Name) to reflect the value of the `ownerName` attribute from the `SmartHouseControl` part. Then we use two radio buttons to reflect the alarm status: The alarm is either on or off. Whenever the door is opened, the Smart House self-activates the alarm. To monitor the lights in the different rooms of the house, we use the Light Control button, and we can specify in which room (represented here by a number) the lights must be turned on.

Table 1 explains in more detail connections we have to create for the SmartHouse monitoring system in Figure 8.

<b>Table 1.</b> SmartHouse Connections			
<b>Key</b>	<b>Source</b>	<b>Target</b>	<b>Description</b>
<b>1</b>	<code>doorOpenedEvent</code>	<code>enable</code>	This event-to-action connection enables the Alarm ON radio button whenever <code>doorOpenedEvent</code> occurs.
<b>2</b>	<code>doorOpenedEvent</code>	<code>activateAlarm</code>	This connection illustrates that a part can be both the source and target of a connection. In this case, the house self-activates the house alarm if <code>doorOpenedEvent</code> occurs.
<b>3</b>	<code>ownerName</code>	<code>text</code>	With this attribute-to-attribute connection, you ensure that the text attribute (that is, the value of the entry field) always reflects the value of the <code>SmartHouseControl</code> 's <code>ownerName</code> data member.
<b>4</b>	<code>buttonClickEvent</code>	<code>switchLightsOn</code>	Because the <code>switchLightsOn</code> action requires a room number parameter, this event-to-action connection is not complete without connection <b>5</b> .
<b>5</b>	<code>value</code>	<code>roomNumber</code>	This connection passes the value of the numeric spin button to connection <b>4</b> as a parameter.

## Visual Builder Editors

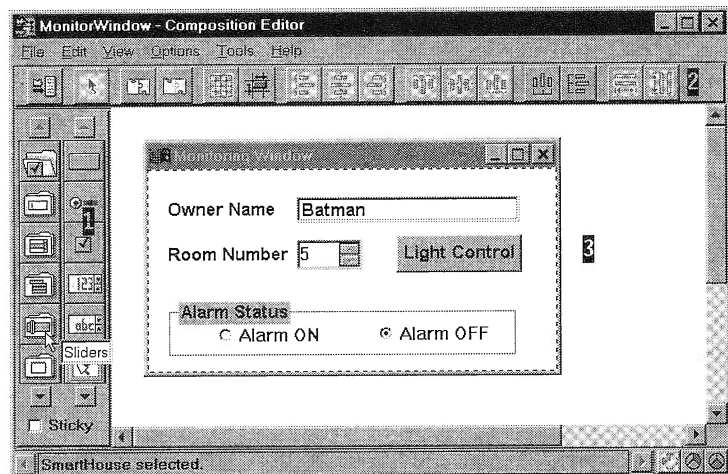
Visual Builder provides three editors that you can use to build your parts.

**The Composition Editor.** With the Composition Editor (Figure 9), you can design the graphical interface of your application, add the nonvisual parts you need for the logic of your application, and make the appropriate connections.

Visual Builder comes with a set of nonvisual and visual parts classified by categories in the parts palette **1** in Figure 9. The base parts are mainly mapped from the User Interface Class Library and the Collection Class Library. The palette can be extended by adding your own categories and primitive or composite parts.

The toolbar **2** provides direct access to a set of tools that you can use to arrange the parts layout on the free-form surface **3**.

To create a new application, just pick up the visual and nonvisual parts you need from the parts palette and drop them onto the free-form surface. Then make the appropriate connections and generate the code.



**Figure 9.** Visual Builder: Composition Editor

Visual Builder can generate the following code:

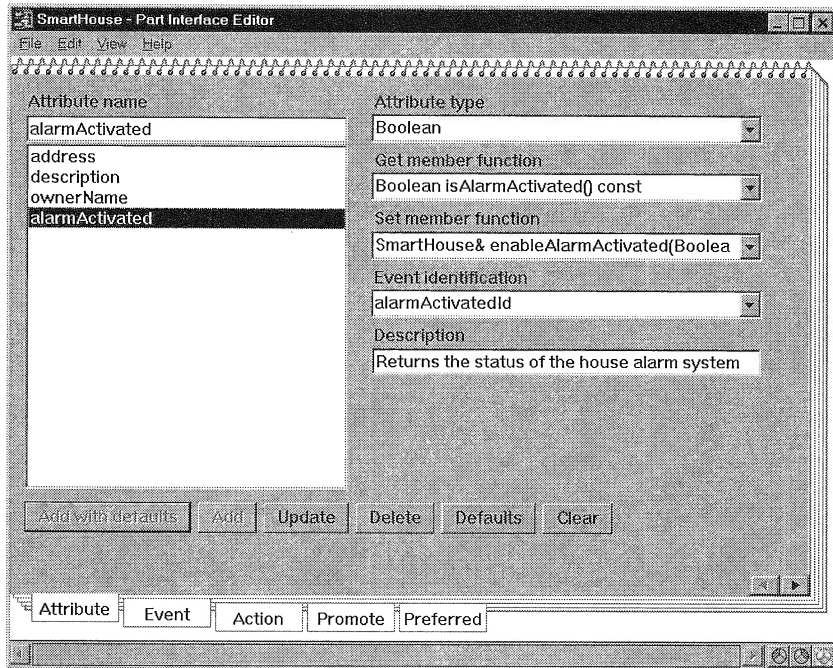
- ☐ Part source, that is, the code for creating the parts and the logic derived from the connections
- ☐ Main source file, that is, a file containing a `main()` entry point, if you want to test your part

- ❑ Make file for building the application (if you are not using Work-Frame and the MakeMake facility)
- ❑ Application resource file (for national language support (NLS))

**The Part Interface Editor.** You can use the Part Interface Editor to create or modify the interface of your parts. With the Part Interface Editor, you can create the attributes, actions, and events related to your part, promote features, and select your preferred features.

### Creating attributes

You create an attribute by entering its name and type. The Part Interface Editor automatically generates the declarations for the attribute accessors (set and get member functions) you need as well as the identification of the event corresponding to the attribute. Visual Builder uses this event to signal any changes to the attribute value. You enter a short description for the attribute. Figure 10 shows an example of using the part interface to create a Boolean attribute.



**Figure 10.** Part Interface Editor: Attribute Creation

## Creating actions

The Part Interface Editor generates a default member function declaration from the action name provided. The tool automatically reflects any changes to the returned type or any addition of parameters to the function call.

## Creating events

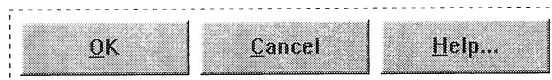
The Part Interface Editor generates a unique event identification from the event name you provide. Visual Builder uses the identification to notify other parts when this event occurs.

An event can have parameters that indicate the name and type of some data corresponding to that event. For example, if your part must read a queue element when receiving an event, the event parameters contain the element address and its type, such as `IString`, so that you can directly access the data with any subsequent query.

## Promoting features

With the promote feature facility, you can provide access to part features when the part is embedded as a subpart within another part. Say you define a defaultButtonsPanel composite part from a simple canvas to which you add three push buttons (*OK*, *Cancel*, *Help*) as shown in Figure 11. If you then reuse the defaultButtonsPanel part in another application, only the attributes, events, and actions of the defaultButtonsPanel base part that is, the canvas are available.

Because you no longer have access to the defaultButtonsPanel subparts, such as the three push buttons, you cannot directly create a connection that would start a specific action when clicking on the **OK** push button. You must *promote* any feature that you want to access from another part that reuses the part.



**Figure 11.** defaultButtonsPanel Composite Part

The Part Interface Editor lets you select a subpart name (such as `OKPushButton`), the feature type (for example, event), and name (for example, `buttonClick-Event`). Visual Builder generates a default name for

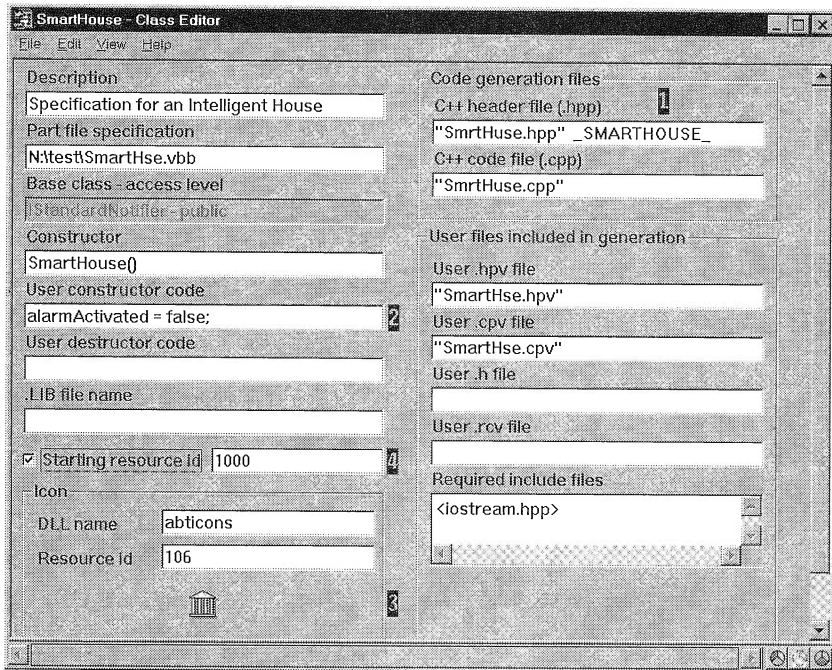
the promoted feature (OKPushButtonButtonClick-Event). You can change this name to anything suitable such as PressedOKEvent.

This name is then added to the list of features for the defaultButtonsPanel composite part.


### Selecting preferred features

You can select preferred features to customize the list of features in the connection menu for each part. The list typically contains the features that you use most often.

**The Class Editor.** With the Class Editor (Figure 12), you can customize code generation parameters. For example, you can change the source and include file names where the generated code is saved **1** or modify the default constructor and destructor code **2**. You can also attach a specific icon **3** to the part. The icon is displayed when you reuse the part in the composition editor.



**Figure 12.** Visual Builder: Class Editor

By default, all graphical resources, such as the label of a push button, are hard-coded in the generated code. If you want those resources to be generated in a separate resource file, you must specify such from the Class Editor, , by selecting the *Starting resource id* check box and providing the entry field with a resource identifier. This feature lets you create applications that are enabled for NLS.

Whenever you create or modify the interface of a part, such as adding an action or attribute definition, the code is generated in user-defined files (with extensions .hvp, .cpv, and .rcv). This code is referred to as the *part features source code*. Unlike other files that Visual Builder generates, user-defined files are not overwritten whenever you generate the part features source code. Rather, the new code is *appended* to the files. Thus, you can modify the generated code without fearing that those changes will be lost at next code generation.

## Accessing DB2 Tables with Data Access Builder



With Data Access Builder, you can graphically map your existing relational database tables or ODBC database sources to an object interface. In a simple case, a database table maps to a class, and a column of the table maps to an attribute of that class. Once you have defined your mapping, Data Access Builder can generate C++ to be used directly into your C++ programs or nonvisual parts to be used in Visual Builder. Moreover, you can take full advantage of the IBM System Object Model (SOM) technology by generating the code in the SOM interface definition language (IDL). (For an introduction to SOM technology refer to “Direct-to-SOM Support” on page 43.)

Data Access Builder comes with a library of classes and parts for database management (connect and disconnect) and transaction services (commit and rollback).

You have the choice between multiple database access methods: You can either access DB2 version 2.1 databases natively through embedded Structured Query Language (SQL) or the DB2 Call Level Interface (CLI), or use the Open Database Connectivity (ODBC) drivers to access multiple databases such as Sybase System 10 or Oracle 7.

### Warning



Although any database product supported by a compatible ODBC 2.0 driver can be reliably accessed via Data Access Builder, only the DB2, Sybase System 10, and Oracle 7 ODBC drivers are officially supported.

## Generated Parts

Let us take a simple example: We create a car table with four attributes, color, license, make, and model, as depicted in Figure 13. If we map this table for use in the Visual Builder and generate the code,

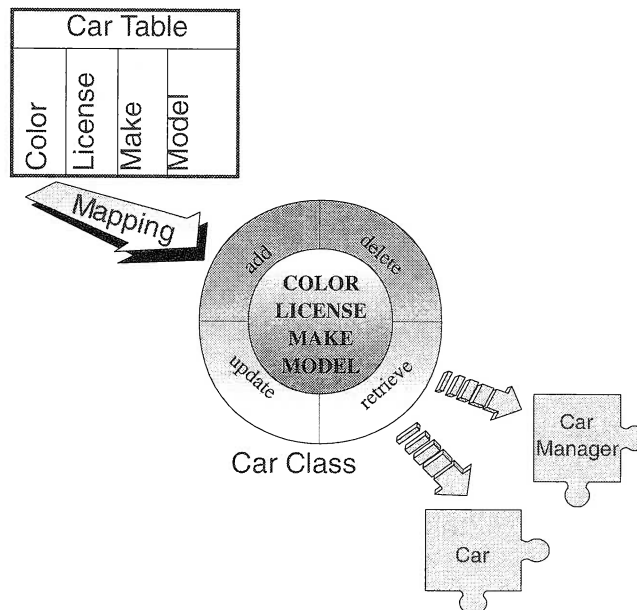
Data Access Builder creates the following main classes:

### ❑ Car

An instance of the Car class maps to a single row of the car table. The Car class provides the member functions for adding, updating, retrieving, and deleting a row of the car table. A Car object is referred to as a *persistent object*.

### ❑ CarManager

The CarManager class provides the services for manipulating a set of car instances. You can use a Car Manager instance to select some rows of your table through an SQL query (select method) or display the complete set of rows (refresh method).



**Figure 13.** Database Access: From Mapping to Parts Generation

Data Access Builder generates other classes that you can use to manage the datastore the table belongs to, as well as some base classes that you can inherit from for your own development. See Chapter 11, “More about Data Access Builder...,” on page 373 for more detail.

Data Access Builder also comes with a library of classes and parts for database management (connect and disconnect) and transaction management (commit and rollback).

## Building from Blocks

Without doubt, one of the greatest advantages of object-oriented programming is class reusability. The IBM Open Class Library provides a comprehensive range of reusable classes from which you can create and manipulate objects. It is supported across many IBM and non-IBM platforms to provide maximum portability of your C++ programs. Most of the VisualAge for C++ tools have been developed by use of the IBM Open Class Library. Notice that the IBM Open Class Library source code is provided with the product.

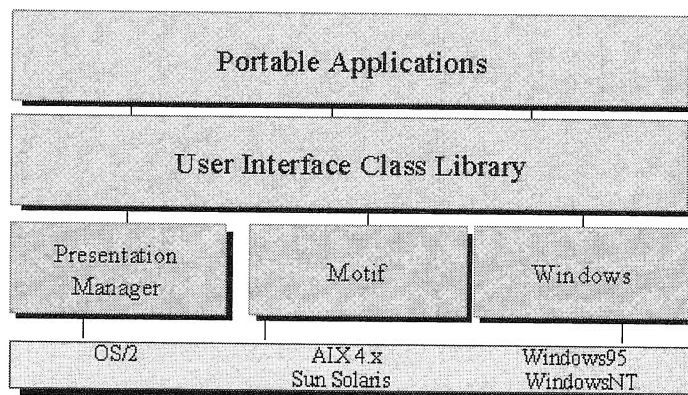
The IBM Open Class Library provides you with more than 500 classes, grouped in the following libraries:

- ☐ User Interface Class Library
- ☐ Collection Class Library
- ☐ Data Access Class Library
- ☐ Application Support Class Library
- ☐ Standard Class Libraries
- ☐ Compound Document Framework

### ***User Interface Class Library***

The User Interface Class library facilitates the development of *portable* applications that have a GUI. It is built as a layer on top of the native window presentation system and encapsulates its concepts in C++ classes (Figure 14).





**Figure 14.** User Interface Class Library Architecture

With a common interface across platforms, you can recompile your code without worrying about the low-level changes of the native operating system. However, some Presentation Manager or Windows features might not be available in another graphical environment such as Motif®, so you must follow some rules to guarantee that your code is fully portable. The documentation precisely identifies portability issues for each class across the available platforms. The User Interface Class Library includes the following elements:

- ☐ Base windows, menus, handlers, events, and help files (display help, define contextual help, and fly-over help)
- ☐ Base controls, such as entry fields, static texts, buttons, and boxes
- ☐ Advanced controls, such as containers, canvases, sliders, notebooks, toolbars, and font and file dialogs
- ☐ Application control classes to manipulate threads, timers, resources, profiles, and the clipboard
- ☐ Dynamic data exchange (DDE) classes for communication between applications on the same machine
- ☐ Direct manipulation classes (drag-and-drop support)
- ☐ 2-D graphics classes for drawing primitives (lines and arcs) as well as support for reading and displaying various graphical formats (available in OS/2 and Windows only)
- ☐ Multimedia classes for control of multimedia devices (available in OS/2 and Windows only)

## ***Collection Class Library***

The Collection Class Library includes a complete set of abstract data types to manipulate such objects as:

- ☐ Bags and sets: unordered collections of elements
- ☐ Sequences: ordered collection of elements
- ☐ Queues and dequeues (double queues)
- ☐ Heaps
- ☐ Stacks
- ☐ Trees

Bags and sets can inherit from various properties such as indexing and sort. As a result, you can use sorted bags or key sets. You can alter queue properties to assign an access priority to added elements.

## ***The Collection Class Smart Guide***

VisualAge for C++ comes with the collection class Smart Guide that lets you choose the appropriate collection for your application and generates sample code using that collection. You may run the SmartGuide either as a novice or as an expert. As an expert, you are expected to know which type of collection you need. As a novice, you are prompted for the following collection properties:

- ☐ Whether the collection elements are ordered or unordered
- ☐ Whether the collection elements are accessible with a key
- ☐ Whether the collection elements should be unique
- ☐ Whether the collection is sorted

Once the type of the needed collection is determined, SmartGuide generates sample code showing how to manipulate the collection. SmartGuide is available from the **Guides→Collections** menu in the LPEX Editor, whenever you edit a C++ file.

## ***Data Access Class Library***

The Data Access Class Library provides classes that you can use to manage connections to a database (authentication, connect, disconnect) as well as transactions (commit, rollback) on the database. It also holds the abstract base classes that Data Access Builder uses in the code generated after mapping tables to classes (refer to “Accessing DB2 Tables with Data Access Builder” on page 33).

## ***Application Support Class Library***

The application support class library provides the classes used most often while developing C++ applications:

- ❑ String manipulation classes provide member functions to edit, compare, convert, format, and test strings.
- ❑ Date and time classes provide member functions to test and compare dates or times, convert date and time formats.
- ❑ Exception classes provide the framework for handling exceptions within the class libraries.
- ❑ Trace classes provide trace facilities to help in debugging code.

### ***Standard Class Libraries***

The Standard Class Libraries consist of the standard I/O stream library for C++ input and output handling and the complex mathematics library for manipulating complex numbers. The UNIX System Laboratories introduced the standard class libraries in the C++ Language System Version 3.0. Since then, the libraries have become a de facto standard and are shipped with all C++ compilers.

### ***The Compound Document Framework***

Compound documents protocols have emerged with the need for creating and saving documents that could embed any type of file format such as sound, text, or images. Obviously, the solution could not be to convert such file format to another but rather to create documents smart enough to save each file separately into its native format into a hierarchically organized file. Object Linking and Embedding (OLE) is one of the compound document protocols available on the market, released by Microsoft in the early 1990s.

The Compound Document Framework (CDF) provides a simplified interface (13 classes) for developing OLE components (such as containers and servers). Using the CDF interface, you considerably simplify the task of coding OLE applications, as most of the default behavior of the OLE components is handled by the framework.

Here are some of the key features of the CDF:

- ❑ OLE2 drag and drop
- ❑ OLE2 cut, copy and paste
- ❑ OLE2 embedding
- ❑ OLE2 linking

See Chapter 13, “More about CDF...,” on page 453 for a detailed description of the OLE2 concepts and programming examples with the CDF.

The CDF framework is the first injection of the Taligent Technology into IBM Open Class and is intended to provide a consistent architecture across OLE and OpenDoc.

OpenDoc is an open, multiplatform architecture for component software. It is supported by the Component Integration Laboratories (CILabs), a nonprofit association of more than 300 members, including leading companies such as Apple, Lotus, Novell, or IBM and the Object Management Group (OMG).

### ***Portability Issues***

The IBM Open Class is clearly intended to be the base for developing portable applications across various platforms. Although source code portability is handled by the IOC library, additional tools are provided to ease the porting task:

- ☐ Resource files converter, which converts resources files from the OS/2 format to the Windows format and vice-versa
- ☐ Icon and Bitmaps converter, which converts icons and bitmaps from the OS/2 format to the Windows format and vice-versa
- ☐ Data Access Builder session converter, which converts DAX sessions files from OS/2 to Windows

## **The Resource Workshop**

Resource Workshop provides tools for manipulating resources, such as menus, help-tables, or graphics. With the Resource Workshop, you can:

- ☐ Create and edit resource files (RC files). Resource Workshop includes editors for dialog boxes, menus, accelerators, graphics, and strings.
- ☐ Edit resources in binary files. If you load an executable or library in Resource Workshop, the tool extracts the resource information and lets you modify it.
- ☐ Create graphics files. With the Resource Workshop Graphic Editor, you can create bitmaps, icons, cursor, and font files.

## **Building Your Application**

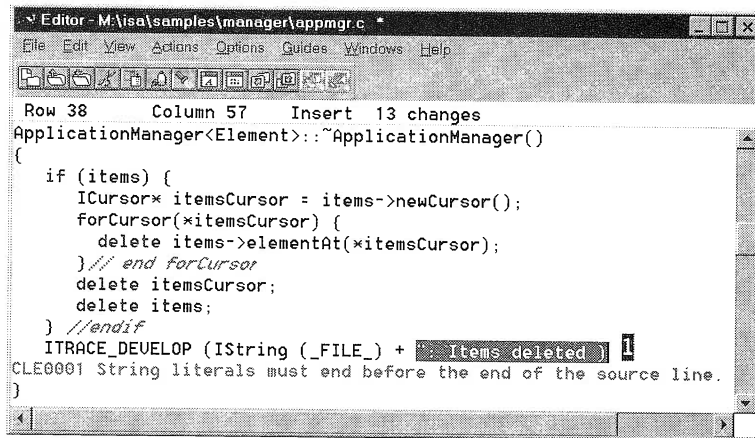
In this section, we introduce you to the VisualAge for C++ Editor and present the main features of the compiler and linker.

## Editing Your Code



The VisualAge Editor, also known as the live parsing extensible (LPEX) editor, is a language-sensitive editor. It reacts to the type of the edited file and uses live parsers for keyword highlighting and code formatting, such as automatic indenting. The LPEX editor dynamically detects errors in your file and does “stupid error checking”; for example, it finds missing closing brackets and detects nested comments. Figure 15 shows an example of C++ source code formatting and dynamic error detection (❗). The LPEX editor is shipped with live parsers for C, C++, FORTRAN, Pascal, COBOL, REXX, and BASIC.

The LPEX editor provides selective views of your code, such as a class definition list, function list, and error list. You can also choose to show only the lines containing a given string. The LPEX editor features all of the functions you would expect from any editor, such as block manipulation, a search facility, a toolbar for fast access to common commands, and multiple views of the same file.



**Figure 15.** LPEX: Source Formatting and Dynamic Error Detection

The LPEX editor is fully customizable. With it you can easily change key assignments, parameters, menus, the toolbar, fonts, and colors and write your own parser or modify existing parsers. You can extend the capabilities of the LPEX editor if you have some REXX and C programming knowledge.

## Compiling

The VisualAge for C++ compiler generates highly optimized code for any Intel® architecture from the i386™ to the Pentium Pro™ and conforms to the major industry standards (ANSI C, ANSI C++ Draft X3J16) to allow you to write portable C and C++ code. It supports the key features of the C++ programming language, including templates and exception handling.

Some of the key features of the VisualAge for C++ compiler are:

- ☐ Precompiled headers
- ☐ Memory management
- ☐ Support for locales
- ☐ Code optimization
- ☐ Run-time type information support
- ☐ Direct-to-SOM support

### ***Precompiled Headers***

With precompiled headers, the compiler does not have to recompile header files each time you change a source file that uses the header files. Precompiled headers improve compile time. The VisualAge for C++ compiler groups precompiled headers in a single file.

### ***Memory Management***

VisualAge for C++ introduces advanced memory management facilities, such as a user-defined pool of memory as well as a debug version of the usual memory allocation functions, for example `malloc` or `free`.

***User Defined Heaps.*** VisualAge for C++ gives you the option to create your own pool of memory, referred to as *heaps*. Using multiple heaps can significantly improve the performance and memory management in your programs. Multiple heaps are particularly useful when you deal with multithreaded applications, or when you manage large objects in memory. Let us consider a simple example: You want to manipulate a large sequence of objects (10000). If you use the regular run-time heap, then you need to walk through the entire sequence, node by node, to explicitly delete each object. If this sequence is created on a separate heap, then you need only destroy the heap itself, which is usually much faster.

***Memory Allocation Debugging.*** Compiling with the `/Tm+` option transparently replaces the call to a memory allocation function for example `malloc`, by its debug version, `_debug_malloc`. Used together with the Debugger, this option enables automatic heap checking each time your program stops on a breakpoint.

For a detailed description of the memory management component in VisualAge for C++, see the *VisualAge for C++ Programming Guide*.

### **Support for Locales**

Locales help you build internationalized applications. They provide a way of changing the behavior of your application according to language and cultural differences, such as character sets and date formats. The VisualAge for C++ compiler provides the facilities to create and manipulate locales in your code. Such facilities include `LOCAL-DEF`, to create a locale object, and `ICONV`, to convert a file from one code set to another. You can either reuse the locale objects supplied with the VisualAge for C++ compiler or create your own.

Support for locales is based on the IEEE POSIX P1003.2 and X/Open Portability Guide (XPG/4) standards. For a detailed description of locales support, refer to the *C/C++ Programming Guide*.

### **Code Optimization**

You can optimize your program by improving its execution speed or decreasing its size. All optimizations that the VisualAge for C++ compiler performs are safe. The VisualAge for C++ compiler uses advanced technologies for code optimization, such as intermediate code linking, global optimizations, or interprocedural optimizations.

**Intermediate Code Linking.** The intermediate code linker combines the intermediate code from several compile units into one compile unit. Thus, because the optimizer does not have to optimize each compilation unit separately, it performs more efficiently for function inlining and global optimizations. The intermediate code linker might also detect errors that would cause unexpected run-time behavior or linker errors such as:

- Redefinition of variables or functions
- Inconsistent declarations or definitions of functions
- Type mismatch between definitions or declarations of the same variable
- Conflicting compiler options

**Global optimizations.** The VisualAge for C++ compiler performs loop analysis, dead code removal, and advanced switch analysis.

**Interprocedural optimizations.** The VisualAge for C++ compiler reorganizes your code for function calls, uses registers for storing variables, and performs instruction scheduling. The built-in functions are optimized according to the processor type.

See the *VisualAge for C++ Programming Guide* for more details on optimization techniques.

### **Run-Time Type Information Support**

Run-Time Type Information (RTTI) is an extension to the C++ language defined in the current C++ draft standard. The C++ language support for RTTI includes:

- ☐ The `dynamic_cast` operator

The `dynamic_cast` operator combines type-checking and casting in a single operation. It verifies the validity of the requested cast and actually does the cast if the operation is valid.

- ☐ The `typeid` operator

The `typeid` operator returns the run-time type of an object. It returns an object of class `type_info`.

- ☐ The `type_info` class

The `type_info` class contains the run-time type information of an object.

VisualAge for C++ provides an extension to RTTI, the `extended_type_info` class. This class provides basic operations for implementing a persistent object store. For more detail on RTTI, see the *VisualAge for C++ Programming Guide*.

### **Direct-to-SOM Support**

Code reusability is one of the great promises of object-oriented programming. The reality, however, is that if you deliver a library of C++ classes in a Windows or OS/2 environment, it will not work in an AIX™ environment unless you recompile and relink the library. Moreover, if you make changes in the library, it is likely that the applications using that library will have to be recompiled. Obviously, delivering the same library in a different programming language, such as Smalltalk, is not a straightforward operation.

SOM addresses these issues and provides an environment where reuse is a reality. SOM clearly separates the interface of a class from its implementation to provide language independence (see Figure 16). With SOM, you can define classes in one programming language and use them in another. You can also update a SOM library without having to recompile the client code (provided that you do not delete any library member).

SOM objects can be shared across processes through the Distributed SOM (DSOM) framework. Processes can be in the same or different systems. They also can run on different platforms. The interprocess communication is totally hidden from the programmer.



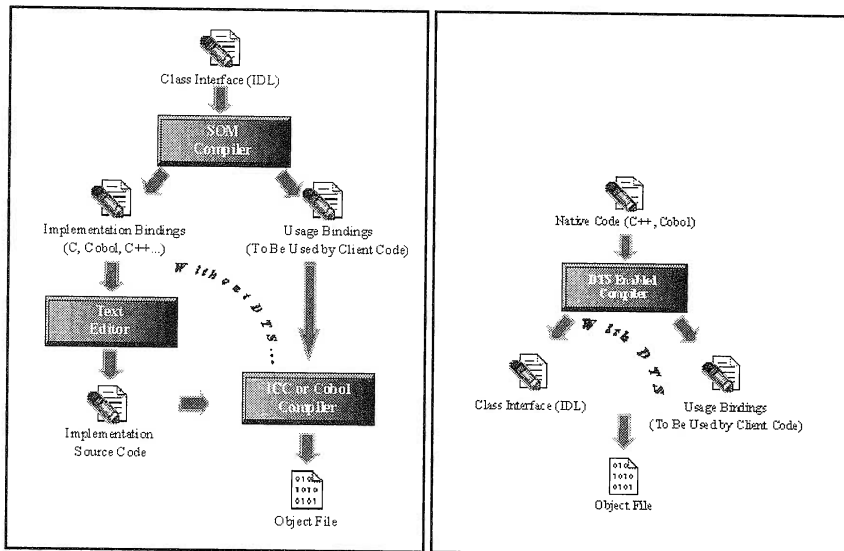
Both SOM and DSOM conform to the Common Object Request Broker Architecture (CORBA) specification of the OMG.

Previously, to create a SOM object, you had to go through the time-consuming process of writing its interface in a neutral language (IDL), generating C++ bindings with the SOM compiler, and compiling the generated C++ source code.

With the Direct-to-SOM (DTS) technology, you can generate a SOM class from a C++ class definition. The compiler also generates the corresponding IDL whenever you want to access that SOM class from another language or use DSOM. Because you are writing C++ directly, you can benefit from the C++ features such as templates, operators, and static members.

Although SOM imposes some restrictions on the C++ syntax, you should be able to convert most of your C++ programs with minimal effort.

See Chapter 12, “More about SOM...,” on page 421 for more details on SOM and Direct-to-SOM, and how to use SOM objects from the Visual Builder.



**Figure 16.** Language-Independent Implementation with SOM

## Linking

Use the VisualAge for C++ linker (or ILINK) to link the object modules created by the compiler. ILINK can produce either executables (.exe) or dynamic link libraries (DLL) in the PE-Image file format.

ILINK accepts the following object files or libraries:

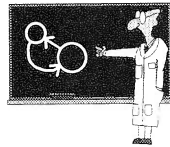
- ☐ Objects and libraries in 32-bit OMF format
- ☐ Objects and libraries in 32-bit COFF format

The VisualAge for C++ linker has the following improved optimization techniques:

### Unreachable functions removal

The unreachable functions removal technique, also referred to as *smart linking*, lets you remove the unreferenced functions in your code or in the libraries with which you are linking. An unreferenced function is any function that you do not call directly in your program or that is not called by one of the functions you call. This optimization technique can significantly reduce the size of your program, and thereby improve its performance. You may want to use this optimization option when your code is tested and stable.

#### Technical Information



If you are linking with a DLL, the unreachable-functions-removal technique will not remove the functions that are exported from that DLL.

### Debugging Information Packing

At the development stage, you can use the debugging information packing option to generate a smaller and therefore faster executable file.

### Code and Data Packing

ILINK *always* packs code and data segments. If you want to specify in your own code to the linker which library to link with to resolve references, you can use the `#pragma library` compiler instruction. For example if you specify `#pragma library "user.lib"` in a `user.hpp` file, then the linker automatically links to `user.lib` to resolve the references declared in `user.hpp`. Using this feature avoids any need to manually specify `user.lib` when you invoke the linker.

## Understanding Your Code

VisualAge for C++ has features that help you understand your code. The class browser graphically displays the structure and hierarchy of your C++ classes. The Debugger helps you understand why your application fails. You can use the Performance Analyzer together with the Debugger for thread interaction analysis, deadlock detection, and performance tuning.

### Browsing Your C++ Hierarchy



Inheritance and therefore reuse are two of the keys to object-oriented programming, but they come at the cost of increased complexity. Finding the right class among the thousands available is often a tricky task.

Browsing helps you analyze and understand which class and its associated member functions can provide the service for which you are looking. With the VisualAge for C++ Browser, you can navigate through the class hierarchy, obtain the interface available to you, locate a function, edit the source files, and access the online help for the IBM Open Class Library classes.

The Browser is particularly useful when you develop a large project with a team of developers. Typically, developers reuse the classes defined in another subsystem and have to know how they can use a particular class and the services offered by that class. Developers also might want to check the impact of changing the prototype of one of their functions that other developers are using.

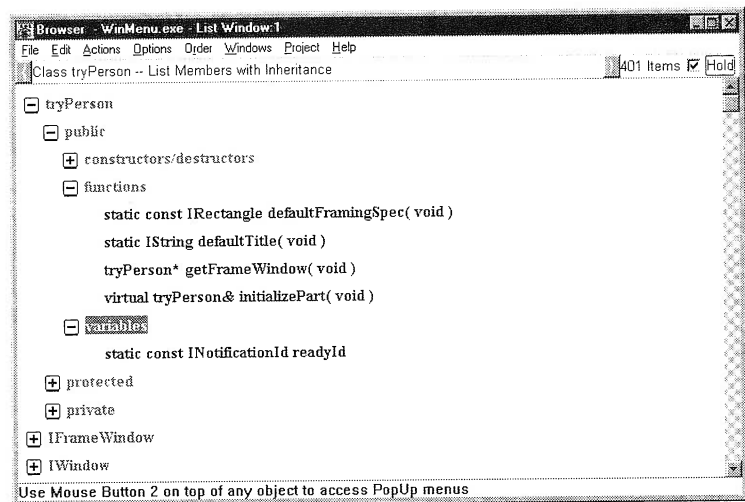
The VisualAge for C++ Browser creates an internal representation of your program in a so-called browser database. The database contains full information about your program if you specified certain compile and link options. However, for those cases when you cannot compile or would like to browse your code before you compile, you can use the QuickBrowse facility. With QuickBrowse, you can generate the minimum amount of information the browser requires to analyze your code. (Some information that would be known only at compile time is not available, such as viewing call chains.) The QuickBrowse facility is typically useful where the project design phase has ended; that is, the definition files (header files) have been completed, but the project has not been implemented.

## Browser Windows

The Browser can display information either as lists or graphs. When you start the Browser, the initial window displays a list of all classes that are defined in your executable file or library, that is, all classes that are defined in the header files you included in your program.

For each class displayed in the initial window, you can access:

- ❑ **List of members with inheritance.** This option displays an incremental list of all members (that is, constructors, destructors, functions, and variables) of the class, as shown in Figure 17. With this list, you develop a good understanding of the complete class interface and have access to all relevant information, such as the online documentation or the header file where these members are implemented. A special notation is used for members that have been generated by the compiler.

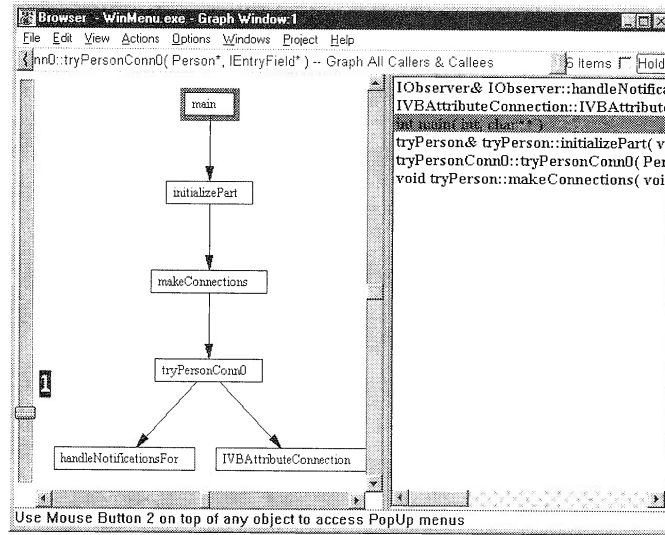


**Figure 17.** Browser List Window: List Members with Inheritance

Class members are usually classified according to the class name, but you can categorize the list by either access (public, protected, private) or type (functions, variables).

For each member function, you can display either the list or graph of caller and callee functions, as well as the list of exceptions that can be thrown from that member function. With the callers and callees graph, you can develop a better understanding of the execution flow of your program, as shown in Figure 18. You can also use this information to measure the impact on the entire application of modifying a function.

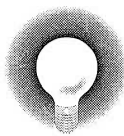
A graph window is divided in two parts: The left side of the window shows the graph itself, and the right side of the window displays the list of objects in the graph. The list of objects can be used to retrieve an object in a graph. It generally displays more information about the object than is displayed on the graph itself, such as the complete definition of a function (Figure 18). The slider on the left side of the graph (1) is used to zoom in on and zoom out from the graph.



**Figure 18.** Browser Graph Window: Graph All Callers and Callees

- ☐ **List of friends or friendships**, that is, the answer to such questions as Whom do I declare to be my friend? and Who declared me as being its friend?
- ☐ **List of implementing files**, that is, a list of header files where the class is declared. With this list you can find the correct header file name to include in your code when you want to use the class.
- ☐ Graph of all base and/or derived classes.

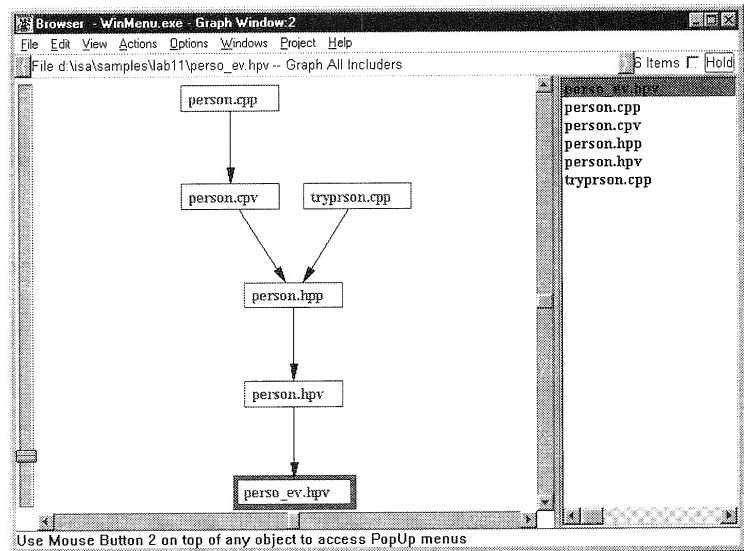
#### Tip



The Browser provides a graph overview window for large graphs. From the overview window, you can select which part of the graph you want to display and zoom in on or out from an area of the graph.

From the initial window, you can also display the list of all files that your executable file or library uses. For each file, you can access the following information:

- ❑ **List of defined objects.** This option displays the list of classes, functions, variables, and types that are defined in the corresponding file.
- ❑ **Graph of all includers and includees** (Figure 19). This option is useful for measuring the impact of modifying a header file in your files hierarchy.



**Figure 19.** Browser Graph Window: Graph All Includers

If you are dealing with a large library, use the search facility for fast access to information. It lets you scan the loaded database and find objects according to simple criteria such as object type, access type, class type, and function type.

### **Visual Builder and Browser Interaction**

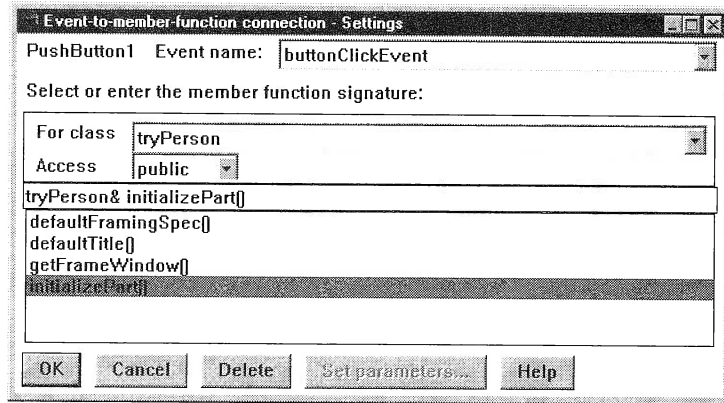
Visual Builder interprets and uses the data stored in a browser database file. From Visual Builder, you can either read a database file or call the QuickBrowse facility (provided that you started Visual Builder from a WorkFrame project).

You can use the browser information to:

- ❑ Create event-to-member function or attribute-to-member function connections. Event-to-member function or attribute-to-member function connections enable you to call a member function of a part whenever the corresponding event occurs (see “Connecting Parts” on page 26).

- ❑ Incorporate existing code with the Part Interface Editor (see Chapter 8, “Creating Nonvisual Parts,” on page 225.)

In both cases, Visual Builder loads the list of function definitions, which you can then reuse. You can directly access those definitions from the various Visual Builder editors. Figure 20 shows the GUI for creating an event-to-member connection after the browser data is loaded.



**Figure 20.** Visual Builder: Creating an Event-to-Member Connection

## Debugging Your Code



The VisualAge for C++ Debugger lets you debug your 32-bit C or C++ code at the source level. You can also use it to debug child processes and SOM objects generated through DTS or the SOM compiler.

The Debugger provides advanced features for breakpoint management, memory management, and functions monitoring.

### ***Breakpoint Management***

You can set a breakpoint at any place in your program where you want to stop execution. The Debugger supports simple breakpoints, such as stopping when a certain line number in a source file is reached. It also supports more complex breakpoints, such as stopping program execution when a certain address in memory is modified or putting conditional breakpoints on variables. (For example, you can stop execution whenever a variable in a loop reaches a given value.)

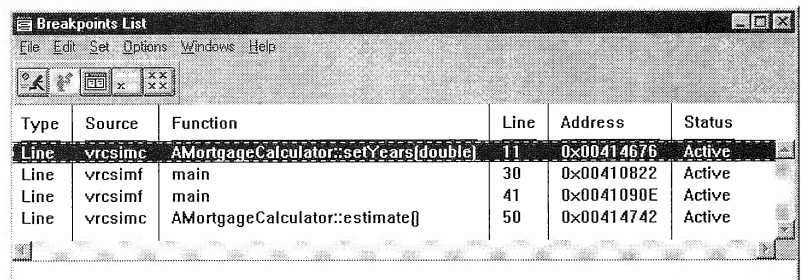
The Debugger supports the following breakpoint types:

**Line** Stops program execution at a specific line number.

- Function** Stops program execution when the first instruction of the corresponding function is called.
- Address** Stops program execution when a specific address is reached.
- Change address** Stops program execution when the contents at a specific address are changed; you can specify either an address or a variable name to set such a breakpoint.
- Load occurrence** Stops program execution when the program loads the specified DLL.

When working with DLLs, you can defer breakpoints so that they are activated only when the corresponding module is actually loaded. This option is supported for line and function breakpoints.

You can manage all active breakpoints from the Breakpoint List window shown in Figure 21.



**Figure 21.** Breakpoint List Window

## Memory Management

With the *check heap when stopping* option, you can perform memory checks each time the program stops executing, for example, when a breakpoint is reached. The check heap facility detects memory block allocation problems such as writing data outside a block segment or freeing the same memory block twice. When the Debugger detects a problem, the program stops executing and displays the exact line where the problem occurred.



## ***Functions Monitoring***

Advanced monitoring functions let you get a complete view of your program's behavior. You can simultaneously track the call stack, storage status, and register values as well as analyze the graphical windows if you are debugging a graphical application.

***Call Stack window.*** The Call Stack window dynamically lists all active functions for a particular thread in the order in which they are called.

***Storage window.*** The Storage window dynamically displays the storage contents and storage address.

***Registers window.*** In the Registers window, you can display and alter the contents of registers.

***Windows Analysis window.*** With the Windows analysis window, you can graphically display the relationship among the windows that are created when you run a Windows application.

## ***Debugging Session Management***

For each application that you debug, you can choose to save the current debugging session. The next time you debug that application, the Debugger tries to restore the saved session, including breakpoints and the various windows that were active when you stopped your previous debugging session.

## ***Debug on Demand***

The debug on demand feature enables the system to start a debugging session whenever an error occurs in a running program. For example, if an unhandled exception occurs, the debugger starts automatically and traces the execution up to the instruction where the application execution failed. You can use the debug on demand feature for any application, even if it has not been compiled to include debugging information.

**Note:** This feature is available only on Windows NT.

## **Performance Analysis**



The Performance Analyzer provides you with facilities to improve application performance or to detect problems at run time that are difficult to find with a traditional Debugger.

When compiling and linking your program with the correct options, you create hooks in your program. The Performance Analyzer uses those hooks to create a trace file. The hooks cause a small monitoring function to be called inside every program's callee function. The monitoring function stamps the event, dumps it into a trace file, and then actually calls the function. Because the monitoring function is called in the program's address space, the trace overhead is minimal. Moreover, all chronological diagrams take the trace overhead into account and are therefore accurate.

### ***Customizing Trace Generation***

The Performance Analyzer views applications as a set of components. A component can be the executable file itself, a DLL, an object file, or even functions. You can influence the size of a trace file by enabling components to be analyzed inside an application.

In you have a multithread program, you can exclude threads that you do not want to trace and select the call depth for those threads. The maximum number of threads you can trace simultaneously is 64.

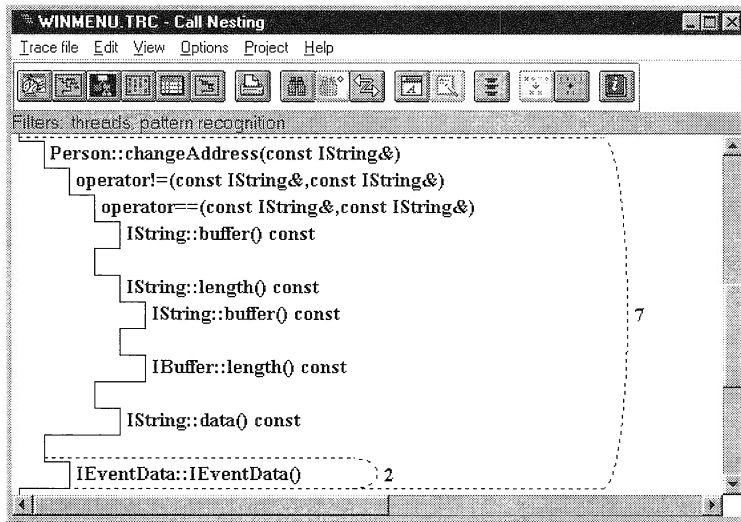
If you need better trace granularity, you can modify your program source. The Performance Analyzer provides two functions calls: `PerfStart()` and `PerfStop()`. You place the calls anywhere in your code to start and stop trace generation.

The `PERF(string)` macro lets you create user events in the trace file. The string you pass as a parameter is dumped into the trace file during program execution. User events in the call nesting, statistics, and time line diagrams appear as diamonds. You also can trace system call events by linking with specific libraries, such as `_kernel.lib` or `_gdi32.lib`.

### ***Performance Analyzer Diagrams***

From the trace file that it creates, the Performance Analyzer provides several diagrams that you can use to time and tune applications, trace thread interactions, find where a program hangs, and detect deadlocks. Filters allow you to temporarily reduce the amount of data displayed in a diagram or graph. The filtering options may vary from one diagram to another but are essentially based on thread numbers and function names.

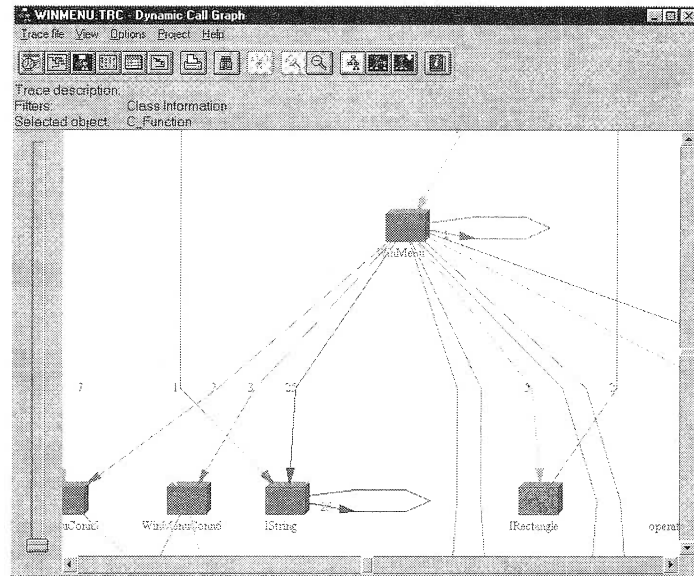
***Call Nesting Diagram.*** The call nesting diagram shows the trace file as a vertical series of function calls and returns. Each thread has its own starting column of functions. Use this diagram to build an understanding of thread interaction in your program—each context switch between threads is represented by a dotted line—and follow the flow of function calls in your application (Figure 22.)



**Figure 22.** Call Nesting Diagram Window

To reduce the amount of data displayed by the call nesting diagram, use the pattern recognition option, which looks at a single thread and finds patterns of calls and returns. The patterns are displayed as curved arcs, and the number of repetitions is indicated on the right-hand side of the corresponding arc, as shown in Figure 22. The pattern recognition view helps you isolate patterns of code that you reuse frequently. Then, for better performance, you can group instructions belonging to a pattern into the same code segment.

**Dynamic Call Graph.** The dynamic call graph is a graphical view of the application execution, where execution components are represented by nodes and relationships between components are represented by arcs. Components can be functions (Figure 23), classes (only if you are analyzing a C++ program), or executables. The dynamic call graph uses color to represent the time spent in each node and the number of calls in an arc. For example, a node colored red indicates that more than one-half of the total execution time was spent in that node. You can also quickly determine the time on stack or execution by looking at the size of a node.



**Figure 23.** Dynamic Call Graph Window: Nodes Of Functions

**Statistics Diagram.** The statistics diagram summarizes all data about functions or executables. Functions can be sorted according to execution time, time on stack, and minimum and maximum time for a call (Figure 24). The trace overhead time is also listed.

WINMENU.TRC - Statistics				
Trace file View Options Project Help				
Summary				
Number of trace buffer flushes: 0				
Total trace time excluding overhead: 21016252 tics or 17.614 seconds				
Trace overhead: 17435 tics or 14.612 milliseconds				
Details				
Class	% Of Execution	Number of Calls	Execution Time	M
Person	91	21	42.620	
ostream	6	3	2.875	
C_Function	3	30	1.177	
IString	0	60	0.176	
IEventData	0	34	0.084	
IBuffer	0	13	0.002	

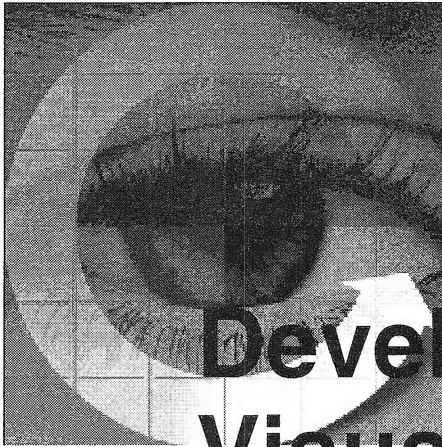
**Figure 24.** Statistics Window

**Execution Density Diagram.** The execution density diagram displays execution time divided into fixed horizontal time slices. If you compare this graph to a table, you can see that each row is a time slice and each column is a function called during the time slice. Color is used to indicate the percentage of the time slice that the function uses.

**Time Line Diagram.** The time line diagram displays the sequence of nested function calls and returns. It is similar to the call nesting diagram, with the exception that time is represented on the diagram. Time is divided into horizontal time slices, and function names appear on the right-hand side of the diagram.

**Windows Correlation.** All chronologically scaled diagrams (time line, call nesting, and execution density) can be correlated according to a particular event or specific point in time. Thus, you can select a time period in the time line diagram and use correlation to get a different view of it in the call nesting diagram.

# Part 2



## Developing with VisualAge for C++

*If you only have a hammer, all your problems look like a nail.*

-Proverb

In Part 2 and 3 you will learn how to build a real-life application, the Visual Realty application, which supports real estate agents who manage properties and customers. A customer of a real estate agency is either a seller who wants to sell property and asks the agency for assistance or a buyer who wants to be the proud owner of new property. So, an agency must keep track of properties that are available for potential buyers and mediate between sellers and buyers. Potential buyers have particular notions of the kind of property they want, so an agent must be able to search for property that matches their notions.

Our goal is not to build a complete application that would solve all of the problems of real estate agents. Rather, we want to illustrate how to apply VMT and use VisualAge for C++ tools to build a good object-oriented software system.

In Part 2 we analyze and design the Visual Realty application to develop static and dynamic models that you will implement with VisualAge for C++ in Part 3.

# 3

## Analysts at Work

*Analysis lets you stay ignorant, but with much more detail.*

-The Financial Procedures Handbook, Apocrypha

To illustrate the process of object-oriented application development we can compare it to the process of writing a novel, because there are as many potential domains to describe in a novel as there are different software domains. We focus on novels that entertain (but not dime novels) because with more complex literature we might lose sight of our actual goal. We do not want to teach dramaturgy or style. Rather, we want to show what the art of writing novels and the art of developing software have in common (from the distorted points of view of computer specialists).

Without playing down the complexity inherent in software development, we want to demonstrate that you can intuitively learn how to apply object-oriented methods. You should not expect, however, to derive benefits from using this approach at the very beginning. It is obvious that the reuse of code in your first project is not of as great importance as it will be in future projects. As usual, you must gather



experience yourself or hire someone who has experience. Likewise, you cannot expect to write a successful novel if you start writing now for the first time.

In the sections that follow, you play the role of a novelist, and we are assigned to be the software developers applying the approach suggested by VMT. We assume that, as a novelist, you do not write as a hobby but want to sell your book in bulk, just as we want to sell our applications to become rich and famous. If you were to write novels just for your private fun, you could write whatever you want without obeying any rules. The same would be valid for us: If we created applications to satisfy our own needs, nobody would blame us if we wrote our code in the old-fashioned, poorly documented, spaghetti-like free-style.

Before we proceed, let us state the big difference between writing novels and developing software applications: Writing novels is an art; developing software is applied science. Therefore, although object-orientation aims to “industrialize” software development some day, assembly-line production is not desirable for the process of writing novels (but we notice such a trend when we look at pulp fiction). We want to be able to enjoy future novels and admire an author’s particular style of writing that fills his or her characters with life and unleashes our imagination. Nevertheless, we want to compare the two processes because software development undeniably involves the developer’s creativity and imagination. In addition, using an unfamiliar field to illustrate the principles the developer should observe is a way of avoiding preconceived notions of how to proceed. The development that implements these principles focuses on applying VMT to the work of a real-estate agency.

Although we cannot partition the preparatory work of an author into analysis and design phases, we discover that preparing to write a novel and the preliminaries of software development have a great deal in common. For example, the preparations take considerably more time than the actual writing or coding. In this chapter we deal with the analysis phase, and in Chapter 4, “Designers at Work,” on page 81, we focus on the design phase.

## Collecting the Material

*If you put yourself in an author’s place, how would you proceed? First of all, you would define your subject, answering the following questions: Would you like to write a historical, contemporary, or science fiction novel? Which contexts should your novel cover? (Contexts include the historical period, the location, and the culture.) Should your novel be a romance, a crime story, a biography, or a drama? If you do not know what you are going to write about, you simply cannot begin, and,*

*if you have only a vague idea of the contents of your book, you should not begin, unless you want to rival some intractable software developers.*

*The decision that has the greatest impact on your novel concerns the subject of the main story. This decision influences your style as well as the selection of the participating characters. You might combine several subjects; for example, if you write a biography, your protagonist probably experiences the vicissitudes of life, which include love, jealousy, passion, grief, comedy, and despair, to name a few.*

*First, you certainly want to create an outline of the main story, that is, the thread of the entire novel. Your publisher probably wants to look at your abstract to decide whether your story is promising or a reject. You should formulate your outline clearly, so that your publisher can follow it. Surely, you will not succeed in formulating your outline on first try. You probably will read it, delete some sentences, rearrange it, or even start again. Your final epitome reveals what your novel essentially is about and serves as a guide, as you further develop your novel.*

*Let us consider what you should include in your outline of the main story: You should define certain constants before you start writing, namely, the location of the action, the historical period, the principal characteristics of the leading roles, and the thread of the plot. To create a successful story you should research the characters of your novel and, if possible, inspect the locations where the events take place.*

## Problem Domain

When we start developing a software application, our first task is to analyze the specific problem domain, distinguish it clearly from the real world, but not worry about any constraints that the implementation environment would impose; in object-oriented application development we call it object-oriented analysis (OOA). The result of our analysis can be directly compared to the outline for your novel, as it answers the “what?” question. In other words, we define the complete function of the application and the system boundaries. Initially, we are probably better off than you are, because our customers tell us what to do; that is, they give us their requirements for the application.

After we have collected all material that is relevant to the system, we must arrange and put the finishing touches on our customer’s requirements. Generally, the specifications lack completeness or exactness and show redundancy. To complete the requirements, we must learn about our customer’s business domain and its terminology. On our first try seldom (between you and me, never) do we succeed in correctly defining what we should develop. Through ongoing communication with our users, however, we gradually acquire both the knowledge and understanding that are essential to fulfilling our task.

The deliverables of the analysis phase help us understand the problem domain as they provide different but complementary views or descriptions of the system that we develop. The deliverables are the complete and exact requirement specifications, the use case model, the sketches or prototypes of the user interfaces for each use case, the class dictionary, the CRC cards for all classes, the static object model, and the dynamic model (refer to *Visual Modeling Technique—Object Technology Using Visual Programming* by D. Tkach et al.).

## Requirement Specifications



We developed seven problem statements when we first thought about the daily work of a real estate agent:

1. Manage buyers
2. Manage sellers
3. Manage properties
4. Manage sale transactions
5. Track earnings
6. Document activities
7. Exchange data with the agency's computer

To keep the implementation simple, we established the following constraints:

- ☐ The application would not cover
  - > Seller management
  - > Sale contracts
  - > Documentation of agent activities
- ☐ The commission for an agent and the down payment for a property would be fixed values computed from commission and down payment rates.
- ☐ Agents and buyers would negotiate directly concerning the sale.

We know that our seven problem statements will not drive us to seize the closest keyboard and start hacking the code, because they are rather vague. So, we must define more precisely what we mean. Preferably, we formulate the definitions together with our customers, because they know best what they want.

In fact, we visited a real estate agency and interviewed two of the agents to learn how they do their job and how computers might help them. Our interview with the agents led us to refine the seven problem statements, as you can see in the 11 requirement specifications listed below. Eleven requirement specifications can in no way cover

the real estate business. If, however, we had to describe everything that a real estate agency requires, this book would be a multivolume encyclopedia, and you would have to read a huge amount of analysis and design scribbles before you could learn VisualAge for C++.

You also can see, in the requirement specifications listed below, the transition from the rather simple, initial problem statements to the elaborate, precisely defined requirement specifications (for example, we explode the first statement into three specifications).

- 1. Record, update, and delete buyer information and preferences.**

Here, the terms *buyer information* and *buyer preferences* must be further defined. As you know from Chapter 1 (“Objects” on page 6), real-world objects have a tremendous number of attributes, but our application must concentrate on those that are most important for the problem domain. (The process whereby one tends to concentrate only on the relevant information for the problem domain is also known as *abstraction*.) So, we decide that buyer information includes the buyer’s name, identification (social security number or driver’s license number), telephone, address, and household income. Buyer preferences, which describe a buyer’s particular notion of a property, include price range, size range (in square feet), number of bedrooms, number of bathrooms, number of stories, type of cooling, and type of heating.

- 2. Search buyers by name.**

It should be possible to search buyers by last name. It should also be possible to do a “pattern” search; for example, look for all buyers whose last names start with “Que.”

- 3. Show buyers who are interested in a selected property.**

From a list of properties, the agent can select one and retrieve the buyers whose preferences match the property characteristics. The buyers should be listed so that the agent can select one to retrieve the detail information.

- 4. Record and update property information.**

Here, registration number, address, area, number of bedrooms, number of bathrooms, number of stories, size (in square feet), type of cooling and type of heating are of interest. Further, a second category of information is relevant for a property, namely, marketing information, such as price, price per square foot, commission for the agent, commission rate, number of days the property has been on the market, down payment rate, down payment value, and status (available, sale pending, or sold). Note that some attributes of the property information match the attributes of the buyer’s preferences. Additional features or information about

other categories that do not fit with the preceding attributes can be entered as textual description. A video that features the property completes the property information.

**5. Show available properties of interest.**

List the properties that have the available status and match some specific criteria, such as area, price range, size range, number of bedrooms, and number of bathrooms. These properties should be listed so that the agent can select one to retrieve the detailed information

**6. Show affordable properties.**

List the properties that have the available status and that a selected buyer can afford according to his or her income. The mortgage calculation should be activated when the price range is not provided in the criteria discussed above. This calculation should remain simple.

**7. Display description or video of a selected property.**

This feature provides potential buyers with a first impression of the property. On the basis of this first impression, they can either take a closer look at the property or avoid it.

**8. Initiate, confirm, or cancel a sale transaction.**

The agent initiates a sale transaction when a buyer decides to buy a desired (and affordable) property. The agency must go through several processes until the buyer is finally the owner of the property. For example, it must determine whether the buyer has liquid assets or is creditworthy, but the Visual Realty application does not cover these processes. Instead it creates a transaction record which states that the property is reserved for the buyer for 10 days from the date of signature. During that period, the property is marked as “sale pending.” The transaction record also states that the buyer has provided a down payment as proof of good faith. If the transaction is canceled, the agency returns the down payment, marks the property “available,” and destroys the transaction record. If the transaction is confirmed, the property is marked “sold,” and the agent’s account is credited with the proper commission.

**9. Search agreement forms by date.**

This function enables the agent to list all properties marked “sale pending” and to examine the associated sale transactions.

**10. Show how much commission the agent earned during the current month.**

We assume that there is a one-to-one relationship between agent and computer, so the application shows the sum of the commissions for the sold properties.

# **11. Receive from the agency's computer or send to the agency's computer properties, customers, and sale transaction data that is relevant for the agent.**

The agency should be aware of its agent's activities, track the status of each property, and acquire information on new prospects.

All information about properties, buyers, and agreement forms is transferred to and from the agency's database. We must consider that a real estate agency usually employs more than one agent. The agency is responsible for maintaining the database, keeping it synchronized, and assigning properties and buyers to agents.

Just as your publisher should agree with your abstract, our customers should understand what we write down and fully agree with the functions of the system. If they do not agree, we must add other specifications or redefine them. At this stage of analysis, we iterate the refinement of the specifications to ensure that the problem definition is accurate.

## **Thread and Subplots**

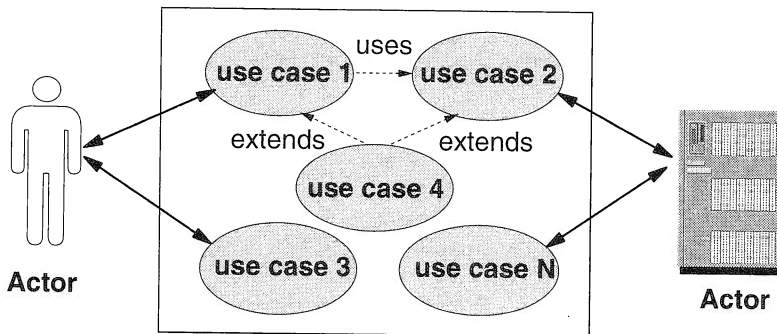
*Your first cut must be appealing enough to convince your publisher of the potential success of the book. Before you deliver your work, you should consult with some reviewers, perhaps a friend or your spouse who can assess whether the flow of the story is comprehensible. Note that you first create the skeleton of the story; only later do you embellish it.*

*In your novel you not only tell the main story, you also delineate several subplots that run concurrently or sequentially. Subplots make the story more interesting, create tension, and make your readers stick to the book. While you sketch the thread of your story, you focus on the plots in which your protagonist is involved. If you wanted to summarize Michael Ende's book, *The Neverending Story*, you could do so very concisely: a boy is absorbed by a book and seeks out a new environment. When he learns that his world is going to vanish, he tries to rescue it with his imagination. (Do not go to your publisher with only two sentences!) Michael Ende used several hundred pages to embroider and embellish his story; that is the actual art of writing.*

We software developers, however, must ensure that our project does not turn out to be a never-ending story. We have already collected and completed the customer requirements. At this stage, we must discuss with our users all services of the system, because our first model must capture all of their functional requirements. We group the requirements into use cases, which are "...behaviorally related sequences of transactions in a dialogue with the system" (*Object-Oriented Software Engineering. A Use Case Driven Approach* by Jacobson et al., p. 127).

## Use Case Model

We write down use cases in the form of a dialog with complete sentences and assign a unique title to each. The dialog represents an interaction between the application that performs the function and the user of the system. According to Jacobson, a use case does not contain conditional branches, that is, each use case describes one distinct sequence of functions. If a function of the use case depends on the successful completion of a preceding function, we must define a second (alternative) use case that covers the event when the preceding function fails. In this case, an “extend” relationship represents the extension between the two use cases. In addition, a use case behavior can be embedded in other use cases, leading to a “use” relationship (Figure 25). Actually, to write the use cases we present the requirement specifications from the points of view of different users. In our case we have only one user: the agent.



**Figure 25.** Use Case Representation

Throughout this book we focus on realizing problem statement 3, that is, *Manage properties*, so the following examples of use cases relate to requirement specifications 4 and 5.

### Record Property Case

Agent: Call the Record property function.

System: Present a form that the agent must fill in to specify all of the required information for a property.

Agent: Fill in the form and click on an OK button or the like.

System: Verify the information and store it if it is correct.

### Property Search Case

Agent: Call the Search property function.

System: Present a form that the agent must fill in to specify the items of interest.

Agent: Fill in the form and start a search.

System: Present the properties that match the specifications.

In the *Property Search* case, the calculation of the mortgage represents an extension of the case:

**Property Search Extended Case**

Agent: Call the Search property function.

System: Present a form that the agent must fill in to specify the items of interest.

Agent: Activate the mortgage calculation.

System: Present the form for the mortgage calculation.

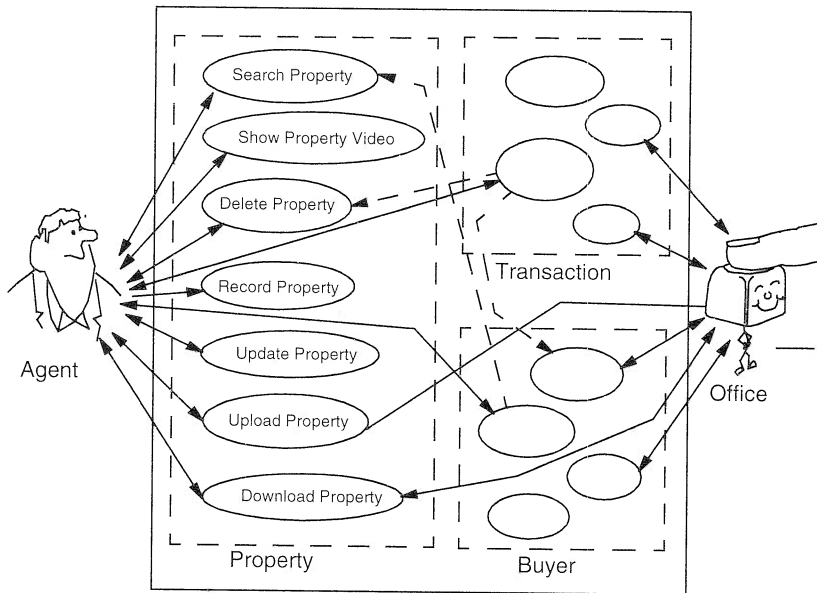
Agent: Fill in the form and run the calculation.

System: Present the properties that match the specifications and the price range.

Jacobson also designates users of the system as actors to emphasize that users play roles. (*Object-Oriented Software Engineering. A Use Case Driven Approach* by Jacobson et al., p. 171). A certain role, or type of user, is assigned to an actor. Usually, our application supports several actors: the “normal” actor, supervisors, and system administrators. The normal actor of the Visual Realty application would be the agent (Figure 26). The supervisor would be the manager of the agency, and the system administrator would be an office worker who cares about the consistency of the property database. Consequently, one person can play several roles. For example, John is a real estate agent who uses the functionality of the property management and, as he sometimes works in the office, of the system administrator, but he is one single user of the system. On the other hand, Mary and Denise, the general managers of the agency, are interested in the supervisor functions only to control John; they are two users playing one role. Generally, the use cases that normal actors activate make up the system’s principal functions, which we focus on during the analysis phase. That is why we interviewed real estate agents and not their managers.

Actors are not necessarily humans; they can be external devices or other computer systems (Figure 26). Each use case is related to at least one actor. The set of all use cases describes the entire function of the system. Together with the set of all actors, it constitutes the use case model. Every actor must be connected to at least one use case, and vice versa; an unconnected actor or use case would be superfluous. In addition, an actor is outside the system; that is, it does not belong to the problem domain, so we do not describe its function in detail.





**Figure 26.** Visual Realty Use Cases

The use case model is fundamental, as it is the collection of all requirements and serves as input for all subsequent models that will be developed, including the final model—the source code. The system's overall function is reflected in the use case model, and we take the use cases to develop the analysis model, the design model, and the source code. Every model can be tested against the use case model for completeness and consistency. If we must change or enhance the application, we do so by first adding or changing one or more use cases and then changing the other models accordingly.

The use case model also helps to maintain the traceability of the system because we know from which use case each component of the later models derives. Remember that traceability is the most important characteristic of our system, as it enables us to make corrections and modifications in a straightforward manner.

## User Interface Prototype

Because we have described the system's functions in the form of use cases, we can envision the user interface of the application prototype. We know which function each actor can invoke, so we can illustrate for potential users what the screen might look like (Figure 27). We can sketch the interface on white paper and then (if we are skilled enough and well equipped), start Visual Builder and paint the screens right

before our customer's eyes. Sometimes, a prototype of the user interface makes the customers bubble over with a wealth of ideas, because they finally see what they can do with computers. In this case, we should stay cool and not promise too much.

The figure shows a prototype of a real estate search application interface, divided into three main sections:

- Property Information:**
  - Real Estate Type:** A list box with options: flat, house, castle.
  - Price Range:** A list box with options: 0 - 50000, 50000 - 100000, 100000 - 150000.
  - Size Range:** A list box with options: 0 - 50, 50 - 100, 100 - 150.
  - Stories:** A text input field.
  - Optional Criteria:** A group box containing three checkboxes: ☐ Parking, ☐ Garden, ☐ Pool.
  - Buttons:** Search, Cancel, Help.
- Video:**
  - A video player showing a house on water.
  - Navigation controls: >>, <<, >, and a stop button.
  - Find:** A button below the video player.
- Marketing Information:**
  - Price:** A text input field.
  - Price/Sqft:** A text input field.
  - Days on Market:** A text input field.
  - Down Payment Rate:** A text input field.
  - Down Payment:** A text input field.
  - Commission Rate:** A text input field.
  - Commission:** A text input field.

**Figure 27.** User Interface Samples

We suggest starting with the most important use case and developing the interface with a top-down approach; that is, begin with the main screen and then define the secondary windows, dialog boxes, and pop-up menus. Again, we come to the best solution after some iteration.

## Defining Roles

*Before you start writing, one of your important decisions is the selection of the characters of your novel. Initially, you only rough out their traits; you refine them later. Barry Morrow, who wrote the screenplay for the movie Rain Man, says: "Creating a character is like shaping a lump of clay, or like whittling a stick." First, the character is rather amorphous; that is, you have only a vague idea of his or her being. Then, the first broad strokes begin to define the character, and you add emotions, values, and attitudes to provide depth. You also can infuse your character with conflicting behaviors to increase the interest and tension of your readers (for example, your character is a tough broker who sentimentally loves his or her children).*

*To create a novel, you can select from several patterns, and the pattern chosen determines the traits of your characters. If you decide to write a romantic novel, you will deal with love, fidelity, affection, sacrifice, and yearning on the one hand, and infidelity, indifference, and heartlessness on the other hand. If you decide to write a crime novel, you will deal with cruelty, loneliness, revenge, as well as heroism, justice, and readiness. You can mix several patterns within your novel to make it more interesting, but the main thread follows one distinct motif.*

## Patterns and Types

The patterns for software systems are also numerous and varied. The *machine control* pattern provides a modest user interface that consists only of some switches, sensors, and digital or analog indicators but must react very quickly to changing process states; sometimes a delay of 1 millisecond can have fatal consequences. The *business graphic* pattern must provide a highly elaborate user interface, but the response time is not so crucial. The *compiler* pattern must hold a lot of information during run time, whereas its user interface is poor, and the response time is of (almost) no importance.

The pattern of the application that we develop determines which object type we must primarily use in our system. Jacobson defines three object types: interface objects, which are responsible for communicating with the world outside our system; entity objects, which mainly store information; and service objects, which control the flow of operation. Highly dialog-oriented applications primarily use interface objects, data-oriented systems primarily use entity objects, and service objects prevail in function-oriented applications.

Do you recognize the correspondence between your characters and our objects, and your characters' traits and our objects' types? As our computer environment is rather unemotional, we deal with only a few object types but many objects. We notice that for each actor (remember that an actor is a user or device outside the system) there must exist at least one interface object, as an actor communicates with our system. Our system also contains at least one entity object, which holds the current state of the process that is running. Service objects are necessary only if we cannot clearly assign a certain function to an entity or interface object or want to guide our user through a sequence of operations.

## Finding Objects

To find the objects of our system as well as their attributes and functions, we must analyze the requirements specifications syntactically and semantically. As a rule of thumb, we can say that nouns in a formulated sentence represent objects, adjectives represent attributes, and verbs represent functions.

So, let us have a look at requirement specification 1: *Record, update, and delete buyer information and preferences*. This specification tells us that we must deal with buyers who have information and preferences, and the user should be able to record, update, and delete that data. If we obey our rule, we would come up with three objects (Buyer, Information, and Preference), but we already have found an exception to the rule, because some nouns can represent attributes. And a Buyer object with information and preference attributes sounds reasonable because information and preference are tightly coupled to the buyer, as we can see from the context of the requirement specification.

Another exception to the rule is synonyms. Often, we encounter synonyms for an object because we interview different people, or we pick up information here and there and write it down, each in our own terms. Thus, during the search for candidate objects and their properties, we must homogenize and structure the requirement specifications and agree on common terms.

Requirement specification 2, *Search buyers by name*, confirms that *name* is an important attribute of the Buyer object and introduces the search function that must be applied for a set of buyers.

Specification 3, *Show buyers who are interested in a selected property*, reveals a new candidate object, namely Property. As the business of real estate agencies is about properties, there is no doubt that property is a real object. In addition, this specification hints that there is a link between the Buyer object and the Property object. The link is based on the property characteristics. We come back to links between objects in “Defining Interactions and Relations” on page 74.

Specifications 4 through 7, *Record and update property information*, *Show available properties of interest*, *Show affordable properties*, and *Display description or video of a selected property*, tell us which attributes are necessary for the Property object and provide information on two additional links between buyer and property based on the buyer’s preferences and income.

Specifications 8 and 9, *Initiate, confirm, or cancel a sale transaction* and *Search agreement forms by date*, introduce the Sale Transaction object with the agreement form and date attributes. Specification 9 states that the agent initiates a sale transaction when a buyer decides to buy an affordable property. It introduces the Agent object and

reveals a link among buyer, property, and agent. The agent manages information on buyers and properties. This leads us to consider two other links among these objects.

Finally, Specifications 10 and 11, *Receive from the agency's computer or send to the agency's computer, properties, customers, and sale transaction data that is relevant for the agent and Show how much commission the agent earned during the current month*, describe three additional service functions.

So, through this syntactical analysis we discover the following objects: Buyer, Property, Sale Transaction, Agent. The next step is to group the objects into classes. If the problem domain is small, as it is in the Visual Realty application, every object typically maps to a class. If we include the management of sellers in the problem domain, we could group the Seller and the Buyer objects into the Customer class.

## Class Dictionary and CRC Cards

*The traits of the characters in your novel are unveiled in the way they think, talk, and behave. It would be worthwhile to establish a fact file for every character in your novel. The fact file would include, for example, the particular behaviors, dreams, and appearance of each character. If you have a talent for drawing, you might even sketch your characters. The goal is to create vivid characters who have consistent traits and with whom your readers can identify. For the most part, the characters impel the action; sometimes, however, some fateful events or coincidences give a fresh impetus to the course of the story.*

As we mentioned earlier, we software analysts must deal with actors who are outside the system; they are users of the system and the main initiators of the flow of the system's functions, but they do not execute the functions themselves. The object-oriented approach assigns the responsibility of executing functions to objects.

### ***Class Dictionary***

As software analysts we also must set up a fact file. Ours is a class dictionary that contains an entry for every class. As you can read in "Classes" on page 7, classes describe the attributes and functions for a certain group of objects. Thus, the entry in the dictionary describes the class responsibilities and the attributes that the class must have. The class dictionary helps us correctly define and develop the classes and serves as a means of communication with our customers. The definitions and responsibilities must be formulated in complete sentences to create a stable base for our object model. We must avoid woolly dictionary entries from the beginning, so that we can rely on our classes and

regard each entry as a binding contract. We can also see whether the existence of a certain class is justified: If we cannot assign any responsibility to a class, we can remove it.

### **CRC Cards**

Wirfs-Brock suggests the use of CRC cards, which can be regarded as extended class dictionary entries because they describe the responsibilities and collaborators of a class. To save some paper or files on our hard disk, we decide to combine the class dictionary and CRC cards so that a CRC card contains not only the name of the class but also its complete description (see Table 2, Table 3, and Table 3). We also suggest omitting the default functions that exist for almost every class: create, delete, and update. However, some classes depend on those functions. For example, as you can see in Table 2 and Table 3, we cannot delete a buyer who initiated a sale transaction that has not been canceled or confirmed.

<b>Table 2.</b> Extended CRC Cards for Buyer	
<b>Description</b>	Person who wants to buy a property
<b>Attributes</b>	ID name telephone address income preferences
<b>Responsibilities</b>	<b>Collaborators</b>
delete	Sale transaction

**Table 3.** Extended CRC Cards for Property

Description	Real estate managed by the agency
Attributes	ID address area number of bedrooms number of bathrooms number of stories size (square feet) cooling type heating type textual description video price price per square foot commission commission rate down payment rate down payment value number of days on the market status
Responsibilities	Collaborators
search	Buyer (preferences)

**Table 4.** Extended CRC Cards for Sale Transaction

Description	Recorded information for the business process sale
Attributes	date agreement form buyer identifier agent identifier property identifier
Responsibilities	Collaborators
initiate cancel confirm	Property Buyer

## Defining Interactions and Relations

*The characters in your novel do not live in isolation; they establish relationships among themselves. Before you start writing, you outline when your characters meet with other characters and determine whether these meetings have any effect on the subsequent action. The relationships may create or resolve conflicts and increase suspense and*

*expectation. Most important, you must manage the sequence of the encounters. Some encounters have more influence on the behavior of a person than others. Some can even completely change a character.*

*You must take into consideration that when you deal with two characters who have a relationship, there are two conflicts. If another character appears—you might think of a love triangle—you must manage six conflicts, as each of the characters has two relationships.*

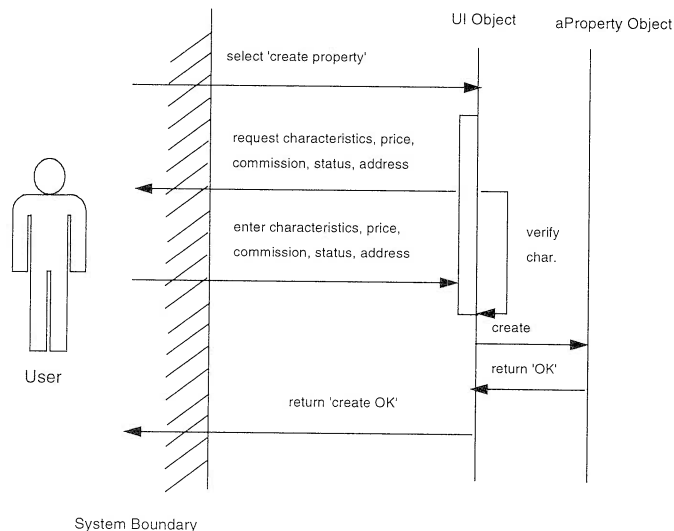
*To keep track of the dynamics, you will want to record encounters and current and future relationships in your fact file. If you must handle a more complex constellation, you will want to make some sketches on a piece of paper that visually document the connections. The relationships are not necessarily of the real world; some could be figments of a character's imagination.*

During the software analysis phase, we build static and dynamic models that illustrate, respectively, the interactions and the relationships or links among objects.

### **Interactions**

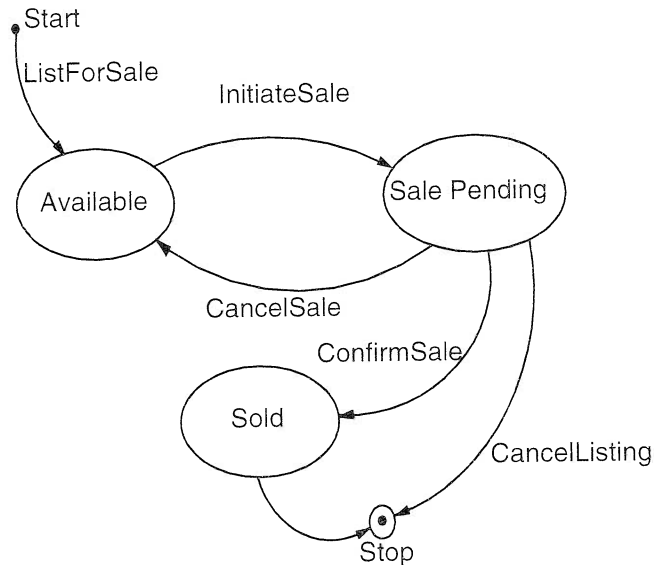
VMT applies two different kinds of diagrams to form the dynamic model: the event-trace diagram and the state-transition diagram. The event-trace diagram describes how the participating objects interact during the execution of a use case. We must create such a diagram for each use case to gain an overall view of the system's functions. We can then summarize all functions or responsibilities with their parameters that are related to a particular object. Because several people might be involved in creating the diagrams, the names of the functions and the number and sequence of their parameters must be homogenized. The diagram (Figure 28) represents objects as vertical bars and the events as a horizontal link between the objects.





**Figure 28.** Event-Trace Diagram for the Record Property Use Case

The state-transition diagram focuses on one object only, regarding it as a finite state machine. It shows every state that the object takes on as the result of an executed function. Each object has an initial state, one or more intermediate states, and, optionally, a final state. Each state is implemented as a distinct value of an attribute of the object. The diagram represents the states as nodes and the event that causes the change of the state as an arc between the original state and the resulting state (Figure 29).

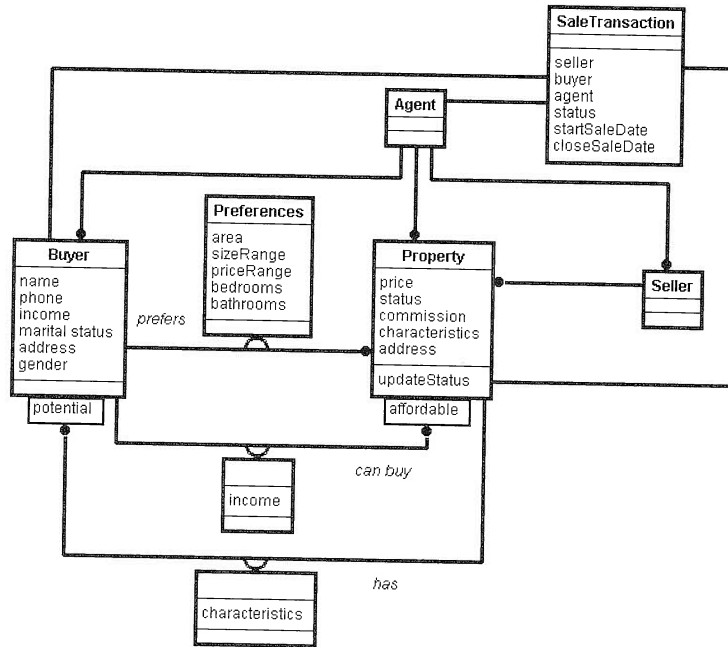


**Figure 29.** State Transition Diagram of Property Status

To avoid having to develop an overwhelming number of diagrams, we are interested only in objects with state changes that are significant for the process flow. A negligible state change would be when the age of a buyer switches from 39 to 40 (although it is of some importance for the person concerned); an important state change would be when the age of a sale agreement form switches from 9 days to 10. We check the state-transition diagram against every event-trace diagram in which our object is involved to ensure completeness and consistency.

### **Relationships**

The static object model, often simply called the “object model,” shows the hierarchy and the coherence of the objects. The coherence is the static relationship, also called the *association*, among the objects. VMT adopts Rumbaugh’s notation to draw the object model and the associations (Figure 30 and Appendix B, “OMT Notation,” on page 511). We assign meaningful names to the relationships so that we can formulate a complete sentence when we take the name of the first object as the sentence’s subject, the name of the relationship as the predicate, and the name of the second object as the sentence’s object. For example: The link between Buyer and Property is established by the buyer’s preference, so we name the link “prefers” and read the link as “Buyer prefers Property.” There are different forms of associations: one-to-one, one-to-many, and many-to-many.



**Figure 30.** Analysis Object Model of the Visual Realty Application

A special relationship between two classes is inheritance (see “Inheritance” on page 8). We could have constructed subclasses of Property, namely, Building and Plot, but we found that the agency does not differentiate between these two kinds of real estate.

In Figure 30, you see six classes and ten associations. The relevant classes are Buyer, Property, and Sale Transaction. The relevant associations are Buyer can buy Property; Buyer prefers Property; Property attracts Buyer; Sale Transaction involves a Buyer, a Property, and an Agent; Seller sells a Property; and Agent manages a portfolio of Properties, Buyers, and Sellers.

#### Read This



For the purpose of completeness, Figure 30 shows the Agent and Seller objects and their links. As mentioned in the introduction, the application does not manage the seller. In addition, the application depicts a fictitious real estate agency where only one agent works. Therefore, the application does not manage the agent information.

## Defining Contexts

*The time during which your story unfolds and where it takes place are of great importance, as they influence the behavior of your characters. For example, if your novel is set in some past era, your characters will necessarily have to speak in a vocabulary that is not contemporary. The time and location of your novel will undoubtedly require that you do some research and look in libraries for relevant documents. The benefit of this bigger effort might be that you attract those readers who are especially interested in learning something about life in other cultures or epochs; you might think of Noah Gordon's novel, *The Physician*, a medieval epoch that aroused his readers' enthusiasm and their thirst for a sequel.*

Here it might be difficult to draw an analogy with OOA. However, when we must analyze an existing application that we want to adopt in our new system, is it not appropriate to research the past? The existing software was developed in the past, so we must consult the documentation (which sometimes exists) and look in the software libraries for the underlying functions of the system. We can also ask former developers how they planned the implementation and maintenance of the system. As for the location of our application, we must consider the hardware and software platforms and many other parameters that relate to the implementation.

OOA excludes reflections on the implementation environment, as we want to achieve an analysis model that is independent of any constraints. Certainly, we must consider the feasibility of our project in terms of the time frame and available financial resources. In addition, we should know something about the user interfaces of the system. Because we and our customers must clearly understand the analysis model, it must not contain descriptions that are too formal or drawings that are too complex.

The primary goal of the analysis model is to use it to communicate with our users, who are not necessarily acquainted with our software terminology. Thus, we must name the objects and their attributes and functions with terms that our users understand. Therefore, we use the same terms that appear in the problem statements, so that sometimes the analysis model is also called the *semantic* model. Actually, it is a symbolic representation of the formulated requirements specifications. We should be able to translate the analysis model into the specifications without difficulty. On the one hand, the model represents the requirements that have been completed and normalized; on the other hand, it serves as a generalized description of the implementation.

As mentioned before, we develop the analysis model without regard to the constraints that the implementation environment would impose on the system. Thus, we can use the same model when we want to implement our application on different platforms. That is why you have not read much about VisualAge for C++ in this chapter. Before we start implementing the application, we must further refine all of the deliverables of the analysis phase to adapt them to the target platform, which in our case is Windows NT. We call this phase of refinement *object-oriented design* (OOD).

# 4

## Designers at Work

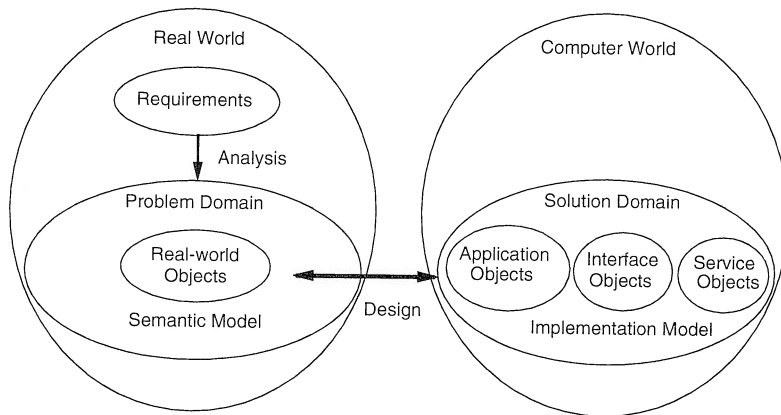
*If you design something that even a fool can use, then only a fool will use it.*

-Corollary to Murphy's Law

The design phase of software development begins when we start thinking about the implementation specifications. We cannot say exactly when the analysis phase ends and the design phase begins, but there is a difference between analysis and design (Figure 31 and Table 5). The analysis model is a conceptual picture of *what* the system provides, whereas the design model is an abstraction of *how* the system is really built. During design, we take a closer look at the details so that we can implement the final solution.

Object-oriented design (OOD) encompasses both system design and object design activities. System design in a client/server environment is a vast and complex topic. Fortunately, the Visual Realty application is a stand-alone software system, although it can generate export files for uploading and downloading data to or from a server.

In this chapter we present the approach we used to design our application with VisualAge for C++. The approach is adapted from VMT.



**Figure 31.** From Analysis to Design

During system design, we decompose the analysis object model into subsystems. The decomposition process is comparable to the process of partitioning the whole application into subapplications. System design also includes the task of choosing a platform and enabling technology as the basis for deriving a high-level architecture. In this chapter, we limit the system design to describing the subsystems of our application (Table 5). Our main focus is on object design with VisualAge for C++.

**Table 5.** Deliverables of Analysis and Design

Analysis Deliverable	Design Deliverable
Use cases, external (user's) view	Use cases, internal (designer's) view
User interface specifications and prototype	Design prototype
Object model	Extension to the object model to include interface and service classes
Class dictionary	Extension to the class dictionary to include interface and service classes
CRC cards	More detailed CRC cards and new CRC cards that describe the classes required for the implementation
Event-trace diagrams (external view)	Event-trace diagrams (internal view)
State-transition diagrams (global level)	State-transition diagrams (detailed level; only for objects with relevant state changes)

Legacy code is an issue when moving to the design phase, because we must integrate it into our application. We must postpone all fine tuning of overall system performance, because we tend to draw incorrect conclusions if the implementation has not been completed. Here the parallel development of a design prototype can help measure run-time behavior, so that we can change our database design as early as possible. The programming language might or might not support object-oriented facilities such as inheritance and polymorphism. It is possible to extend each programming language to an object-oriented language. However, if this is a first project, the task is formidable, as coding rules must be defined and functions that simulate object-orientation must be implemented. In our case, however, we need not worry because VisualAge for C++ provides those functions.

## System Design

*You have already sketched the thread of the plot, and now you define further subplots that support the main flow of action. You structure the overall action by partitioning it. Some authors partition their novels into parts, they further divide the parts into chapters, and sometimes they even subdivide the chapters into subchapters. Optionally, you can assign a title to each chapter and part, so that readers can learn the structure of the action by reading the chapter and part titles. You can also regard a chapter as entirely self-contained, because the actions taking place in it are so closely related that it can exist independent of the book. Indeed, you can assign the writing of such chapters to another author.*

System design is the design of a high-level architecture for the proposed solution. It includes a definition of the major system building blocks and their high-level connectivity. It also includes an application architecture that organizes the solution in subsystems.

Our main tasks during the system design stage are to:

- ☐ Partition the object model into subsystems
- ☐ Map subsystems to VisualAge for C++ subapplications
- ☐ Select the implementing platform
- ☐ Define data placement and data processing

### Partition Object Model into Subsystems

We partition the system into two or more subsystems, mainly to reduce complexity. You can compare a subsystem to a self-contained chapter of your novel; we also want our subsystem to be self-contained. We choose objects for a subsystem that are closely coupled by relationships or that form a functional unit (for example, if one object



is a collaborator for a function of another object). We can read the functional units from the CRC cards, and we can see the relationships when we look at the object model. As a rule of thumb, we can assign the use cases belonging to one actor to one subsystem. If one actor can invoke several use cases, we then focus on the objects. If some of the use cases employ a certain group of objects, we build a subsystem including that group of objects. The objects inside a subsystem are only loosely related to objects outside the subsystem. If some single objects cannot be grouped into any subsystem—objects that are responsible for exception handling, for example—we treat each of them as a special subsystem.

There is one difference when we compare a subsystem to a self-contained chapter in your novel: Some subsystems are regarded as an extension of the base system and are sold separately as service packs. Your readers would be very unhappy if they had to pay extra for the last chapter of your novel.

We can split the effort of system design by developing and implementing subsystems simultaneously with more than one team. As the objects in different subsystems are only loosely coupled, the message flow between subsystems is much simpler than the message flow within subsystems, and the teams can thus work rather independently. We can further partition the subsystems into more low-level subsystems.

We partition the Visual Realty application into three major subsystems:

- ❑ **Property:** This subsystem provides agents with all of the functions they need to manage their property portfolios.
- ❑ **Buyer:** This subsystem provides agents with all of the functions they need to manage their buyer portfolios.
- ❑ **Sale transaction:** This subsystem provides agents with all of the functions they need to manage the sale process.

## Map Subsystems to VisualAge for C++ Subapplications

With VisualAge for C++ we can make a relatively smooth transition from subsystems to subapplications, as the application design tool of VisualAge for C++, the Visual Builder, isolates subapplications in so-called Visual Builder Binary (VBB) files.

We can consider the following VBB files:

- VRPROP.VBB contains the parts required for the property subsystem.
- VRBUY.VBB contains the parts required for the buyer subsystem.
- VRSale.VBB contains the parts required for the sale transaction subsystem.
- VRSERV.VBB contains the parts required to perform basic services (mortgage calculation, upload and download data).
- VRComm.VBB contains common parts (for example, logon view and address view) that can be reused several times in the application.

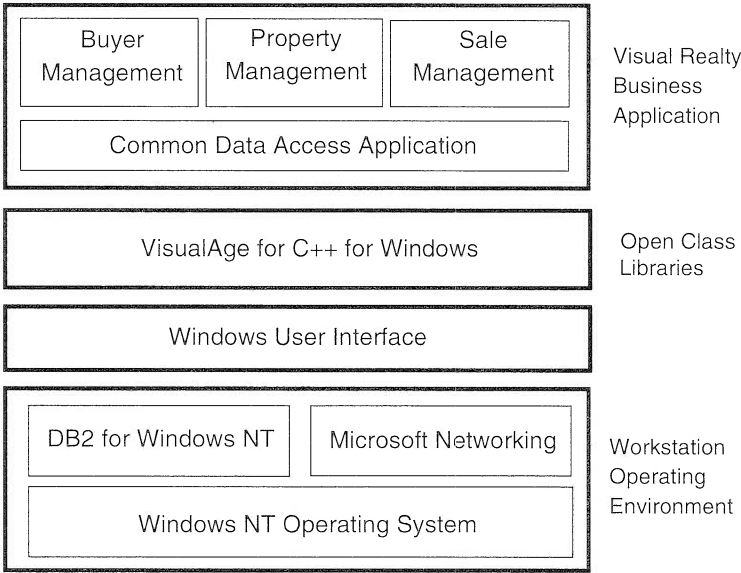
## Select the Implementing Platform

The analysis model is the ideal model, but our computer world is not ideal. The components of the actual implementation environment, namely, the programming language, operating system, database management system (DBMS), networking system, and other software packages, impose some constraints. The design model must include these components, but the goal is to make our problem domain objects as independent of the platform particulars as possible, particularly if we intend to implement our product on different platforms. Therefore, we create new service objects that serve as an intermediate layer between the functions of the actual platform and our business objects.

An example is database access: If a function of a business object directly invokes an SQL query, we will have to change and recompile the object when we (or the customer) bring in another DBMS. You could justly retort that a new DBMS always involves a change, whether it concerns the business object or the service object, so we would have to expend the same amount of effort no matter which object we change. We agree, for the current implementation. If we think of reusability and maintenance, however, we prefer the solution with the intermediate object, because then our core objects, which deal with the business logic, are independent of the underlying DBMS. If we are lucky and land a new customer who needs the same application, we can stay cool and ask: "What would you like, DB2 or Poet?" Even if the customer wants a brand-new, fuzzy-logic-oriented, PCM-CIA-based, 128-bit DBMS, we will still deliver the same core objects. (All we have to do is buy a bigger coffeemaker and create an incentive for our programmers, who will implement the new service object.)

To make the application workable and manageable, we must select the system platform or infrastructure. In our case, the implementing platform is assumed to be a DB2 for Windows NT local area network (LAN) environment. (The Visual Realty application has also been fully

tested with DB2 Windows 95 single-user.) The enabling technology and component selections for our system building blocks were predetermined by both the available supporting platform for VisualAge for C++ when we wrote this book and the needs of our application (Figure 32).



**Figure 32.** Visual Realty System Platform

The Visual Realty application is simple. It uses a stand-alone database so that the agent can carry his or her laptop when visiting customers. The agent can update the local database and generate export files. Back at the agency, the agent can import the files to the central database.

## Define Data Placement and Data Processing

The initial decisions for data placement and data processing are made during system design and can be reassessed during the object design stage. These decisions would include whether the data should be stored in a local or remote database.

As mentioned previously, the application described in the following chapters uses a stand-alone database and the data are located on the same machine. In Part 3 of the book, we describe some advanced features provided by the Data Access Builder that allow you to distribute your data across the network using the DB2 client application enabler and ODBC supports.

## Refine Contexts

*You have already decided when and where your story takes place. Now you must describe the context more distinctly, like a camera that zooms in on a scene. You place your characters in a period setting or describe the particulars of the landscape. Now is the time to consider how you want to embellish your novel. Do you want to describe a sunset, or perhaps a rebellion of gnomes? Remember that the environment you describe influences the personality, behavior, and speech of your characters. All elements must fit together well so that your readers can immerse themselves in your fictional world.*

## Object Design

The object design phase includes a refinement and a fleshing out of the object details. Of course, the level of detail for the object descriptions can vary.

Our main tasks during the object design phase are to:

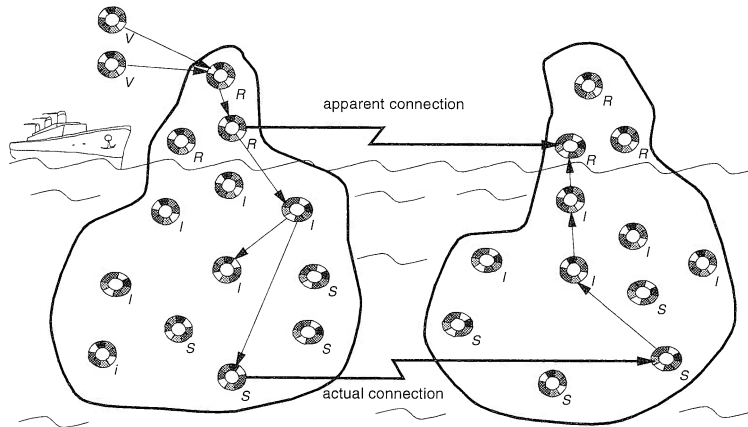
- ☐ Design the solution domain classes
- ☐ Design the nonvisual parts
- ☐ Design the GUI with the visual parts
- ☐ Design the persistent data

## Design the Solution Domain Classes

The set of classes that make up an application is usually much larger than the set of classes identified during the analysis phase. As we consider the circumstances of the implementation environment, we find objects that we must include in our model. New objects emerge that help implement the services or serve as an interface between the application and the world outside the system, namely, the users or their connected devices. The initial semantic application classes identified in the analysis object model represent only the “core” business behavior of the application. Other solution domain classes must be designed to provide the concrete functions of the application. Interface classes that represent the user interface and service classes that provide service functions such as data input validation or database access are some examples of additional classes required for the implementation of the application.

We can compare the visibility of the analysis model with the top of an iceberg, the main part of which is hidden under water. Our design model reveals the hidden objects (Figure 33). To maintain traceability, we translate every object into a part that we later implement, using VisualAge for C++, in a separate source module. The solution domain

classes and the services class are mapped to nonvisual parts, whereas Interface classes are mapped to visual parts. The core is and should remain the analysis model. The supporting objects are settled around this core, providing the services that are necessary to embed the system in the implementation environment.



**Figure 33.** Design Model: Reveal Hidden Objects

The design of solution domain classes is iterative, as is everything else in object-oriented development. For example, in the Visual Realty application, we must have a service class to access the database and control database access with a login procedure. This need does not come out at the start of the design phase but after several iterations. As shown in Chapter 6, “Mapping Relational Tables Using Data Access Builder,” on page 131, this service is provided by the IDatas-tore part.

In the detailed design object model, objects are represented either as nonvisual or as visual parts to facilitate a straightforward implementation that uses the Composition Editor of VisualAge for C++. To avoid complexity, we do not draw all of the parts. However, we show them in the different event-trace diagrams for the subsystem we describe.

## Design the Nonvisual Parts

Once we have determined the required solution domain classes, we are ready to flesh out their details and map them to nonvisual parts. To help us in this task, we use the object model, the prototype interface, and the event-trace diagrams for the use-case scenarios for each subsystem. The user interfaces enable us to flesh out some base

attributes or derived attributes. The event-trace diagrams detail the message flow between the objects and reveal missing objects or missing methods.

Let us take our CRC cards containing the class attributes and responsibilities. We complete the cards by adding the data types of the attributes, naming the functions that carry out the responsibilities, and defining the number, sequence, and data types of the parameters of those functions.

The Visual Realty application is datacentric. Thus, most of the nonvisual parts we use are mapped from database tables. Nevertheless, when designing a nonvisual part, you must pay attention to the following issues:

- ❑ Choosing the right data structures to support object relationships. VisualAge for C++ provides you with a set of predefined data structures (see Chapter 2). The choice is dictated by semantic considerations. For example, an association with multiplicity 0 or 1 between Class A and Class B is represented by an attribute in each part interface referencing Class A or Class B. An association with multiplicity 1-m between Class A and Class B is represented by an attribute of type *collection of Class B* in Class A and by an attribute of type *Class A* in Class B.
- ❑ Designing derived attribute policies. Indeed, it is often useful to make a distinction between primitive attributes that cannot be derived from other attributes and attributes that can be derived (for example, price per square foot is an attribute that can be derived from price and size).
- ❑ Designing the data integrity policy. The programmer devotes a great deal of time to building controls on user data. Two main alternatives can be devised:
  - Target objects expect that data is valid when passed to them and the sender object is in charge of checking the data.
  - Target objects verify data when they are asked to modify their state.

As you will see in “Event Handler” on page 237, we adopt the first alternative, hooking event handlers to some entry fields at the view level.

## Design the GUI with the Visual Parts

During the analysis stage, we have defined the analysis object model with only nonvisual parts. In the design and detailed design phases, we must define the visual parts.

We recommend a bottom-up approach to achieve part reusability. You start at the bottom of the class hierarchy and build one or more elementary visual parts for each nonvisual part that we have created. (Visual parts are also referred to as *views* in this book.) You build the elementary visual parts by using primitive GUI controls (such as entry fields, list boxes, and push buttons). You can then aggregate these views to build more complex views, which represent the final assembly of the final end-user interface.

## Design the Persistent Data

We use Data Access Builder to map our database tables to nonvisual parts. Because our application is data centric, our Data Access Builder parts play the role of the business nonvisual parts. However, some nonvisual parts, such as MarketingInfo, are built to hold some logic that is relevant to the agency's business rules. The relationships between the parts are simulated by means of joins with primary and foreign keys. For example, the address information of a property is not located in the property table; it is in a separate table. When the property information is accessed, the address information is retrieved at the same time and displayed in the property view.

In the next section, we select the Property subsystem and explain how to refine its design model to come up with a detailed model that is ready to be implemented by use of VisualAge for C++.

## Refining the Design Model

*You know exactly when and where your story takes place and you know your characters. Now the time has come to make things happen. Relationships and encounters imbue your story with decisive impulses. You have already defined the relationships in your outline and fact files, but they are static. Now you must make them dynamic. Encounters, meetings, appointments, dates, and coincidences, fictional or real, enliven relationships and further develop the action. They increase the tension, but you can also use them to slow down the main action and let your readers take a few deep breaths.*

*One of the tasks that will challenge you the most is creating vivid and clear dialog. As in real life, not all is said that is thought. Sometimes you have to let your reader read between the lines.*

Now the hard work starts for us: We must draw a detailed dynamic model for all of our use cases, but now we must also consider all participating objects and all objects that we may discover later. During the analysis phase, we described only those use cases that an actor initiates directly, and we reflected only the business objects. Now we

must also consider those use cases that are created “under the covers” and reflect the objects that are invisible to the user. We can see on the event-trace diagrams how the objects interact with one another. Initially, actors create events when they give any input. (According to Jacobson, *Object-Oriented Software Engineering. A Use Case Driven Approach* by I. Jacobson et al. p. 147, they “send stimuli”.) These events are partially handled by the interface objects, as the actor should receive an immediate feedback, but the interface object passes the event to another object that is responsible for carrying out the actor’s request. Events, in fact, are function calls to other objects, which in turn should provide information or carry out a service. We must define the names and parameters of every event with meaningful names to facilitate maintenance and reuse.

Each use case has a normal course and several alternative courses that handle exceptions. Sometimes we find abstract use cases that are comparable to subprocedures. (An abstract use case is a sequence of operations that can be reused in one or more real use cases.) When we build a big application, several designers develop the dynamic model simultaneously. Thus, we must homogenize the model (that is, we must find the smallest number of methods, detect methods with common behavior, and give them a unique name).

After we develop all event-trace diagrams for one object, we can start implementing the object. The diagrams give a complete picture of the object’s interfaces. We also can draw each object’s state-transition diagram to show which method has an impact on the object’s state. As a rule of thumb, we can say that each object maps to one class. If the object plays several roles, however, we should map it to several classes. Object-oriented languages help us to seamlessly translate the dynamic model into source code. As the translation can be done in a straightforward manner, a code generator can be applied here. Humans must still make the final refinements.

From the different views we have sketched in the user interface prototyping phase (see Figure 27 on page 69, for example), we can envision the visual parts we need to implement all of the use cases. Furthermore, from the event-trace diagrams we can chain these different views and discover some nonvisual parts that we need to complete the process. For example, to search for a property by its characteristics, users access a primary window where they can choose the search option. This option brings them to a secondary window where they select several characteristics of “their” property. If needed, they can require a mortgage calculation to approximate the house price range they can afford. Then they launch the search. The result is displayed by means of a table.

For our purposes, we consider the three main functions of the property subsystem: property retrieving, property creation, and property update.



As mentioned, refining the design object model is an iterative process that involves the existing use cases, their corresponding event-trace diagrams, and the design object model itself. On the basis of the part available in the implementation tool and the detailed description of each use case, we evaluate the parts required to complete our process. We apply this refinement process for each event-trace diagram and then modify the design object model.

In the sections that follow, we illustrate the refinement process for the property retrieval, property creation, and property update use cases. We start from a first cut of the property subsystem object model (Figure 34), which is based on the following assumptions:

- ❑ The property information is divided into separate objects that are stored in separate relational tables.
- ❑ A Property object is represented on the screen as a notebook.
- ❑ A distinctive object, PropertyManager, is required to manage a set of properties. In effect, requirement specifications 5 and 6 (see “Requirement Specifications” on page 62) imply the need for a property set structure that must be managed somehow.
- ❑ A set of properties is shown to the user as a container control that holds one or several container objects. A container object is a particular view of a Property object (the other view is the notebook view).

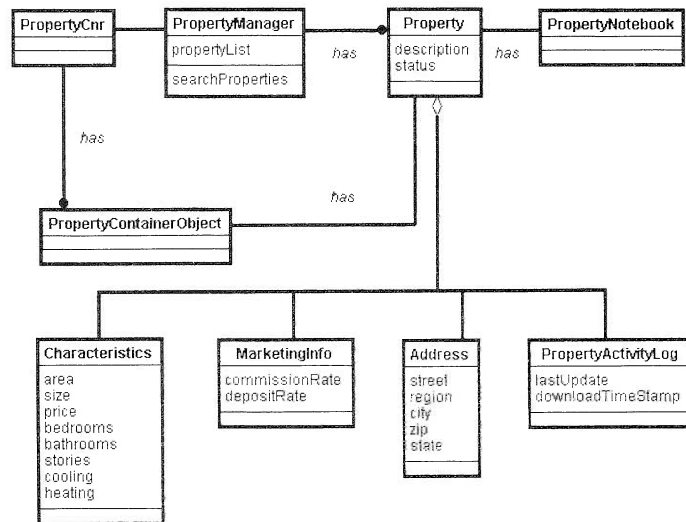


Figure 34. Design Object Model of the Property Subsystem: First Cut

The links between the different objects represent association relationships. The *has* link expresses an association between two objects. For example, the Property and Address objects are associated to express that a property *has* an address. The link between PropertyManager and Property indicates that a PropertyManager manages one-to-many properties.

## Refining the Property Retrieval Scenario

The extended use case corresponding to a property search is expressed as follows:

When the user selects the search option, he or she is prompted to enter his or her search criteria, such as price range, size range, area, number of bedrooms, number of bathrooms, number of stories, type of cooling, and type of heating. Optionally, the user can activate a mortgage calculation to determine what price range is affordable. Once the user provides the information, the user activates the search. The properties that match the criteria are displayed in tabular form.

In this use case scenario, we feel the need for some extra views to complete the process. We can then refine the first event-trace diagram, adding three views that help the user navigate through the application:

- PropertyManagementView is a primary window that enables users to choose the search option.
- PropertySearchParameterView is a secondary window that enables users to enter their search criteria.
- MortgageSimulationView is a secondary window that enables users to enter information in order to estimate a mortgage amount.
- PropertySearchResultView is a window that displays a table of properties that match the users' criteria.

In addition, we need to introduce another object, MortgageCalculator, which provides the calculation for the mortgage and which is used by MortgageSimulationView.

At the detailed design stage, we must take into account the presentation characteristics of the target platform. In our case, the application runs on a stand-alone system under Windows NT. Taking advantage of the Windows user interface controls, we use the detailed view representation of a container to display the properties as a table.

When the user selects the *search* option from PropertyManagementView, a secondary window is created: PropertySearchParameterView (Figure 35). This secondary window prompts the user to enter his or her criteria. Optionally, the user can activate the mortgage calculation to determine an affordable price range for the property. In this case SimulMortgageView prompts the user to enter additional information such as the annual income or the interest rate of the loan to be contracted. This information is transmitted to MortgageCalculator which returns the highest mortgage the buyer can afford. This mortgage value determines the price range. The price range and the other criteria are sent as a clause to PropertySearchResultView. This clause is used by PropertyManager to extract the matching properties. Then, PropertyManager refreshes the property container, which is displayed by PropertySearchResultView. (Although not shown on Figure 35, PropertyManager is embedded in PropertySearchResultView to refresh the property container.) Thus, PropertySearchParameterView and PropertySearchResultView are associated by the clause. We can then refine the design object model by adding these two classes, which are associated by a link attribute clause.

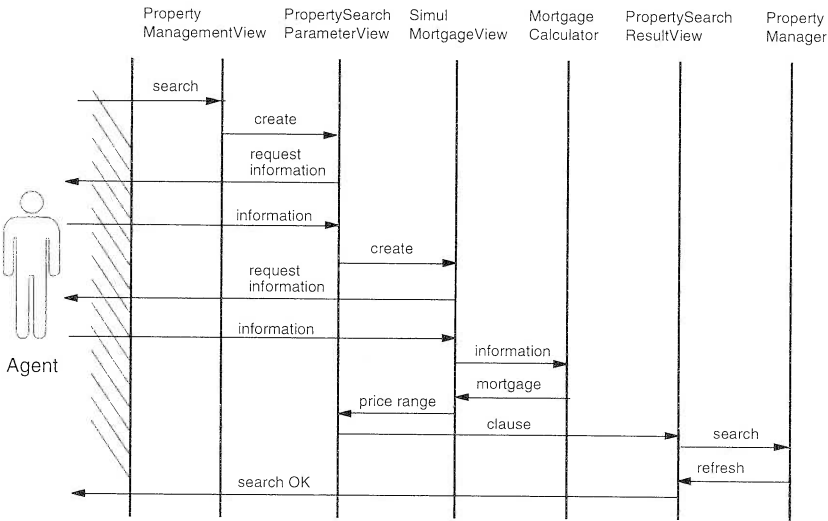


Figure 35. Event-Trace Diagram for the Property Search Use Case

Furthermore, PropertyManagementView is the first panel that is displayed to the user when he or she accesses the Property subsystem. It is linked to PropertySearchParameterView by a *use* relationship called *create*. The create relationship states that the user can access the *Search* option from the PropertyManagementView. In the same way, a SimulMortgageView object is created by PropertySearchParameterView when the user activates the mortgage calculation. During the user interface prototyping phase, it is decided that the user

must close the secondary window to access the primary window. (In Windows it is said that the secondary is shown modally.) For this reason, the relationship clause is a one-to-one association.

Remember that in the first design stage we had to introduce the *PropertyManager* class to manage a set of properties. The visual representation of the class was a container control, and the representation of each property was a container object control.

In the second cut (Figure 36), we can aggregate the *PropertyManager* and the *PropertyCnr* classes to the *PropertySearchResultView*. *PropertyManager* and *PropertyCnr* are associated by the *show* relationship, which states that “*PropertyCnr* shows the contents of the *Property* list managed by *PropertyManager*.” In addition, *PropertyCnr* holds one-to-many *PropertyContainerObjects*, each of which is associated with a *Property* instance by the *has* link attribute.

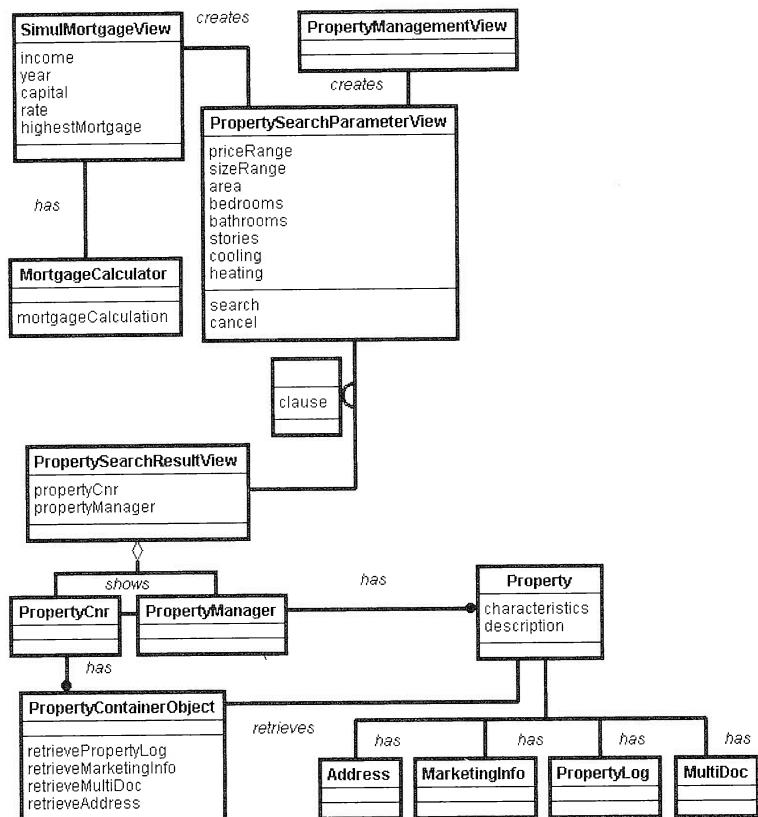


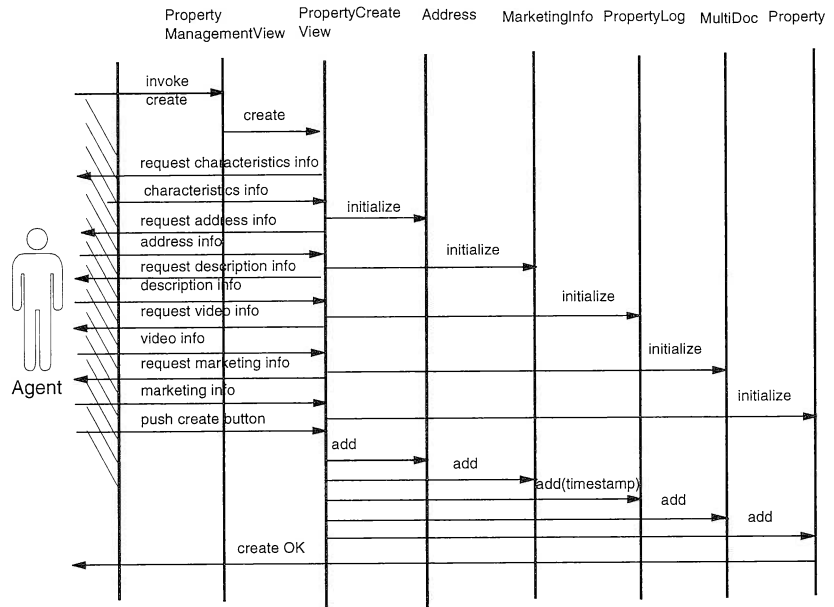
Figure 36. Design Object Model of the Property Subsystem: Second Cut

## Refining the Property Creation Scenario

From `PropertyManagementView`, the agent can select the *create* option to record a new property in the portfolio. To provide the agent with a way of entering the property information, we must again define one extra view, `PropertyCreateView`. According to the first design object model of the property subsystem, this view presents the property information as a notebook. To reuse this notebook in other scenarios (see “Refining the Property Update Scenario” on page 99), we decide to make this notebook a separate view: `PropertyView`. Thus, `PropertyCreateView` contains `PropertyView`.

The user enters the property information in `PropertyView` and creates the property in the portfolio by selecting the *create* option of `PropertyCreateView` (Figure 37). The create order is sent to the `Property` part in charge of creating a new instance. It is also dispatched to the other nonvisual parts to create an instance of each respective part:

- `Address` holds the location information.
- `MarketingInfo` holds the marketing information, such as the price per square foot or agent commission.
- `PropertyLog` holds two time stamps: one for the creation and one for the last update. The time stamps are used during database upload and download.
- `MultiDoc` holds the path name and file name of the video file.



**Figure 37.** Event-Trace Diagram for the Property Creation Use Case

For the third cut, we refine the design object model as follows (Figure 38):

- ❑ PropertyCreateView is added and linked to PropertyManagement-View by a *creates* relationship.
- ❑ PropertyView is added and linked to the PropertyCreateView by a *containment* relationship.
- ❑ Each notebook page is added as a separate view and aggregated with PropertyView to make up the notebook.

The PropertyView notebook consists of five pages:

- ❑ Characteristics page displays the property characteristics (area, size, price, bedrooms, bathrooms, stories, cooling and heating).
- ❑ Address page contains all of the location information about the property.
- ❑ Description page describes the environment of the property.
- ❑ Video page allows the user to watch a video of the property.

- ❑ Marketing page displays some marketing information (for example, price per square foot, sale commission) correlated with one another.

Figure 38 shows the four components of Property: Address, MarketingInfo, PropertyLog, and MultiDoc. Each component is associated with its corresponding page. The Characteristics page contains the descriptive information contained by the Property object itself. The user may ask for some adjustment in the user interface. For example, one page is added for the description of the property, although this attribute is part of the Property object. This is an implementation choice; it does not involve any changes to the model itself.

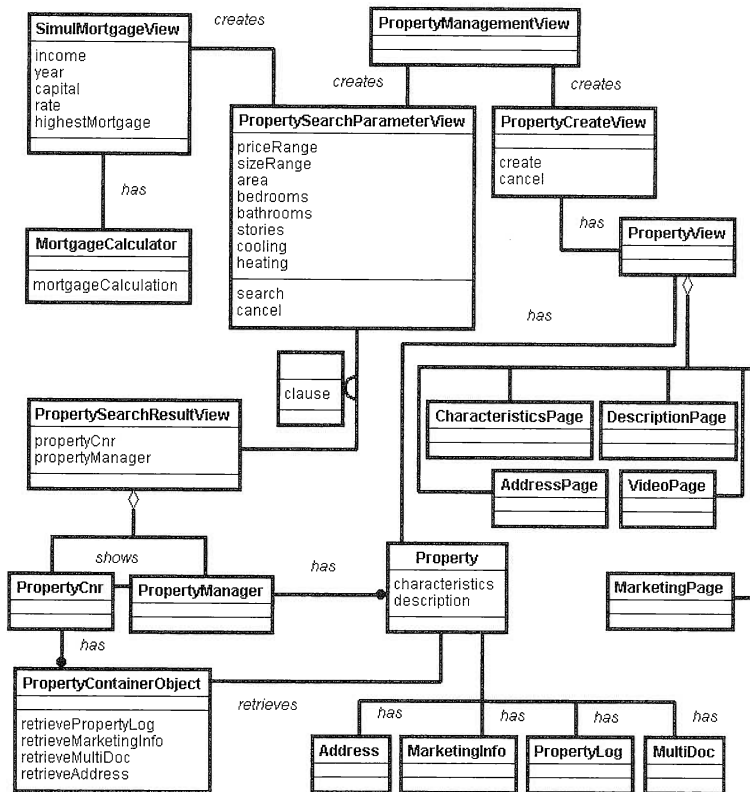
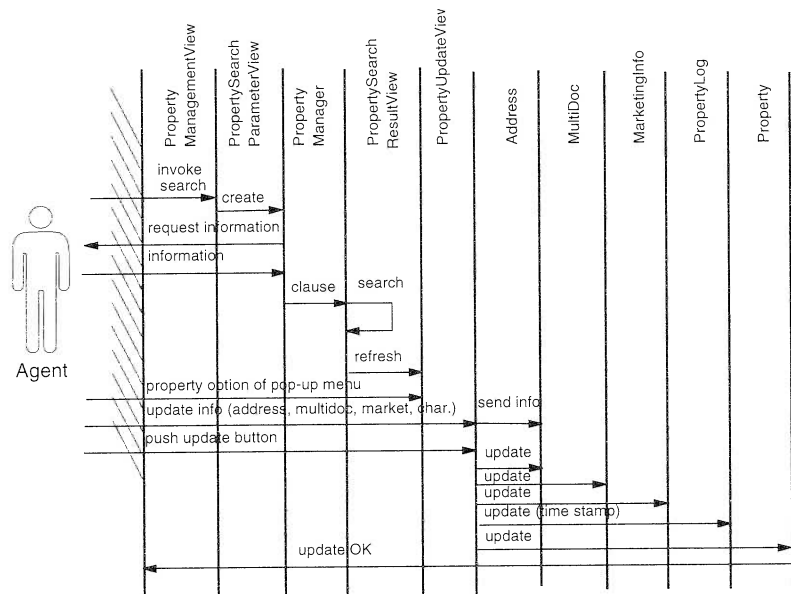


Figure 38. Design Object Model of the Property Subsystem: Third Cut

## Refining the Property Update Scenario

A property search results in a set of matching properties that are displayed in a container. The users can update a property of their choice by selecting the *update* option from the PropertyCnr pop-up menu. Thus, from PropertySearchResultView, users must access an extra view, PropertyUpdateView, which displays the property information and enables them to update it if necessary. This view is based on the PropertyView notebook, which we reuse (Figure 39).



**Figure 39.** Event-Trace Diagram for the Property Update Use Case



PropertyContainerObject is linked to PropertyUpdateView by the *creates* association. PropertyUpdateView holds PropertyView with the *has* containment relationship (Figure 40).

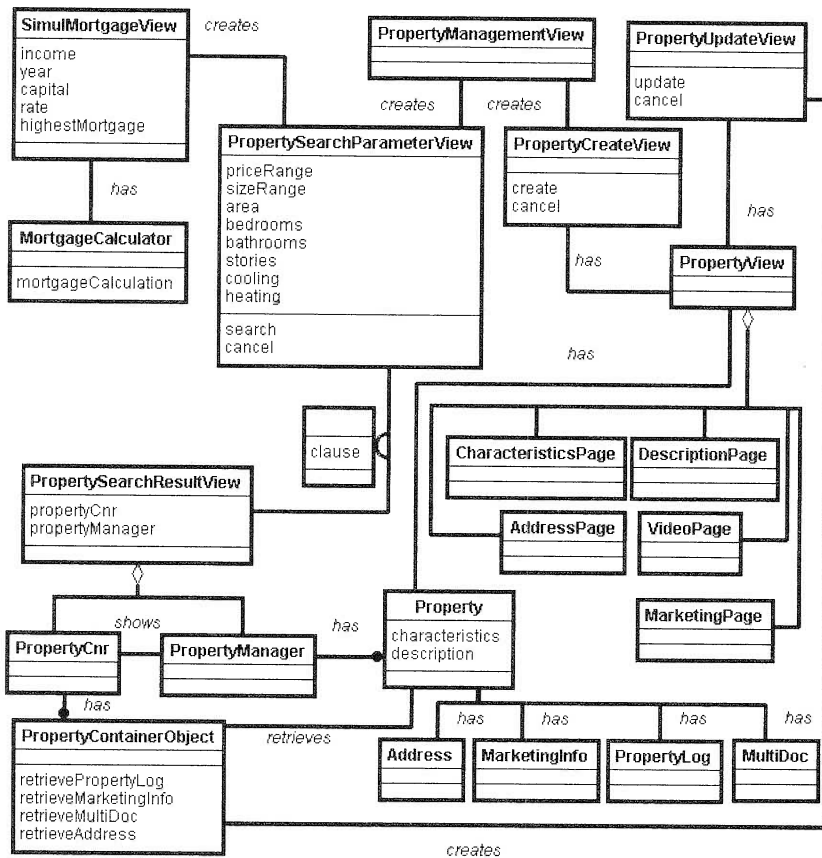


Figure 40. Design Object Model of the Property Subsystem: Fourth Cut

## Refining Roles

*The fact file that you keep for each character who plays a major or minor role in your novel describes his or her main traits rather superficially. Now, you must elaborate on those and add others to make the character come alive. Your reader should almost hear your characters breathing. You reveal what the characters think, describe their inner monologs, and depict in detail how they react to certain situations. Your readers eventually come to know a certain character as well as another character in your novel knows him or her.*

*At this stage, you may introduce new characters to illustrate something about the background of your protagonist. Because some of these new characters play a supporting role only, you might describe them shallowly. Perhaps they appear on a few pages and are never mentioned again. Other new characters may have a greater impact on the course of the story, and therefore you describe them more thoroughly. For example, if your novel is about a baseball trainer, you will describe how he handles his team members, thus revealing his ability or inability to do his job. Suppose that one of the team members has some personal problems, which the trainer helps him resolve. The team member with problems has a more important role in your novel than his comrades have, so you describe his traits and experiences, but not those of the rest of the team. In other words, you focus on characters who carry the plot.*



## Part 3



# Building the Visual Realty Application

By now you must be eager to see how to build the Visual Realty application, so we give you the opportunity to do so in Part 3. Of course, we do not fully detail the implementation of the entire application; rather, we provide you with the keys to build it yourself.

To help you avoid some traps and pitfalls, we focus the development on one subsystem, the Property subsystem, whose main functions are creating, updating, deleting, and retrieving properties. All of those functions take full advantage of the user interface and data access parts that come with the IBM Open Class Library, shipped with VisualAge for C++.

From the detailed design object model, each class is mapped to its corresponding part in Visual Builder: The views are mapped to visual parts, and the business classes are mapped to nonvisual parts.

In the chapters that follow, we show you how to build the subsystem in five steps:

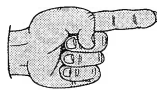
1. You set up your development environment and configure the Visual Realty application project, using WorkFrame.
2. You map relational tables to nonvisual parts, using Data Access Builder. Later, you use these parts with Visual Builder to enable persistence in the application. We provide you with hints and tips that help you design a good mapping and use the classes necessary to interact with the database.
3. You build the different visual parts that are required in the Property subsystem, using Visual Builder. We show you how to build simple visual parts by assembling primitive parts. (See “Using Visual Builder” on page 24. The primitive parts are also called *controls*.) Then we teach you how to reuse the simple parts to build more complex composite parts. We provide you with some design tips to improve the look and feel of your application and explain how to use such complex controls as containers, notebooks, viewports, and multicell canvases.
4. Although most of the nonvisual parts are generated by Data Access Builder and can be used as is, we show you how to design your own nonvisual parts and take advantage of the notification framework.
5. Using specific connections, you assemble all of your parts. We justify the need for variables, explain the consequences of using the promoting part feature, and demonstrate how to take advantage of dynamic memory allocation by means of the factory part.

# 5

## Setting Up the Development Environment

In this chapter we present step-by-step instructions for creating and configuring your development environment with WorkFrame and Project Smarts. You will use the WorkFrame Build facility to create the appropriate make files and build the application's executable files and libraries. WorkFrame concepts are introduced in Chapter 2, "Getting Started in a VisualAge for C++ Environment," on page 19. If you are not familiar with VisualAge for C++, you should read Chapter 2 first.

### Read This



Throughout this chapter, we use the terms *classes* and *actions*. We do not use them in the sense of classes from the object-oriented world or Visual Builder actions. Rather, we use them in the special sense of WorkFrame class and action definitions.

We assume that the VisualAge for C++ product as well as the Visual Realty application are installed on your D: drive. For portability, all file names have been built according to the file allocation table (FAT) format. If you are using a system that understands long names such as the Windows NT™ file system (NTFS), you may want to change the file names to be more self-explanatory.

In the design phase, we identified the different subsystems that make up the Visual Realty application. If you apply the subsystem organization of the application to the WorkFrame environment, you can map each subsystem to a project. In this chapter, we show you how to customize the Property subsystem projects and subprojects.

The Property subsystem manages the creating, deleting, updating, and retrieving properties in the Visual Realty application. Persistent data of the Property subsystem is managed by using DB2 Embedded SQL and accessed from Visual Builder through the parts generated by Data Access Builder.

See Chapter 11, “More about Data Access Builder...,” on page 373 for detailed information on your environment setup if you are using ODBC or DB2 CLI to build the Visual Realty application.

## WorkFrame Projects Organization

To organize the development environment for the Property subsystem, you first have to identify the project parts, that is, the data files that are required to build the subsystem. The Property subsystem data files can be classified as follows:

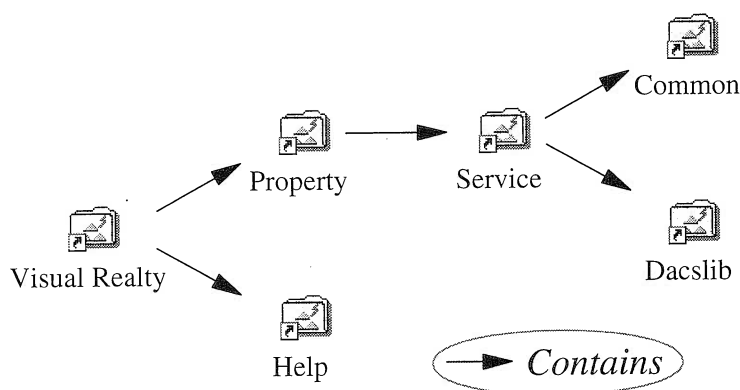
- ☐ Nonvisual and visual parts that are used to build the Property subsystem; Visual Builder can generate those parts, or you can create them.
- ☐ Nonvisual parts created by Data Access Builder that are reused by Visual Builder to manage the Property subsystem persistent data.
- ☐ Visual Realty application common data, that is, the data that all subsystems might use, such as some dialog windows or keyboard handlers.
- ☐ Service subsystem data.

Once you have identified the project parts, you then have to identify the dependencies among those parts to determine the project hierarchy. In the Visual Realty application, the Visual Builder data depends on the Data Access Builder data; you cannot compile the Visual Builder files if the Data Access Builder files have not been generated. In such cases, you must define the project that manages the Data Access Builder parts as a subproject of the project that manages the

Visual Builder parts. This project is called Dacslib (for DACS library). The Property subsystem also uses the Common and Service project data, which implies that the Common and Service projects need to be built first for the Property project to be completed.

Therefore, you must configure the Property project such that it accesses the header files and libraries generated in the Dacslib, Common, and Service projects.

You also have to create a main project that manages the application *main()* entry point, the Property subproject and a Help subproject for the application. The main project is called *Visual Realty*. Figure 41 depicts the Visual Realty application project organization.



**Figure 41.** Project Organization for the Visual Realty Application

## File Organization

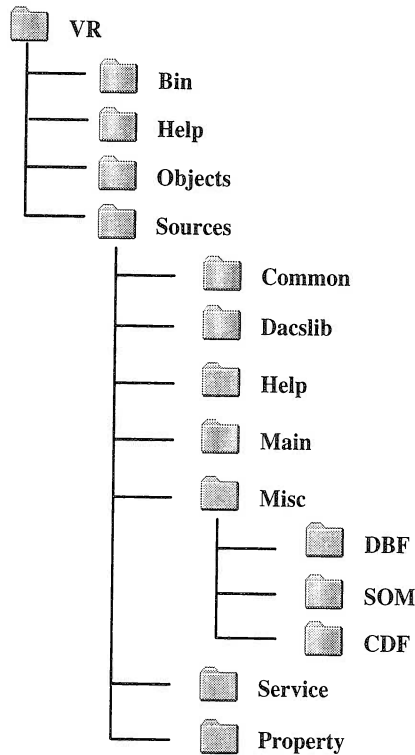
The project organization of the various subsystems does not necessarily reflect the organization of the physical files managed by the Visual Realty projects.

The Visual Realty files are organized as follows:

- ❑ All source files are grouped under a **Sources** directory, which contains one subdirectory per subsystem. It also contains a **Misc** directory where the SOM-related and CDF-related files are stored.
- ❑ All binary files, including executables, libraries, and export files are located under a **Bin** directory.
- ❑ All objects files generated by the compiler are stored in a separate **Objects** directory.



Figure 42 summarizes the file organization.



**Figure 42.** File Organization for the Visual Realty Application .

#### Portability



If you intend to port your code to another platform, you might want to use this information: A WorkFrame project can manage several directories, such as:

D:\VR\SOURCES\MYFILES ( 1)  
D:\VR\SOURCES\MYFILES\GENERATE ( 2)

You can tell WorkFrame to generate all files in Directory 2, which is then referred to as the *working directory*. While working with Visual Builder, put only your VBB files and your user-written code files in Directory 1, and put all of generated code (which is platform-dependent) in Directory 2. If you have to deliver your application on several platforms, you just create one “code generation” subdirectory per platform and change the WorkFrame project settings accordingly.

In the sections that follow, we describe the steps to:

1. Associate IWP Project Files with the WorkFrame facility.
2. Create and customize the DACS library (Dacslib) project for the Property subsystem.
3. Create and customize the Common, Property, Service, and Help subprojects, as well as the Visual Realty main project, with Project Smarts.

## Associating IWP Files with WorkFrame

You must associate WorkFrame projects with the WorkFrame application if you want to be able to open a WorkFrame project by double-clicking on the project icon. A WorkFrame project is represented by a pair of files: An IWP file contains the settings of the project, while the IWO file contains such user-defined settings as compile or link options. Both files must have the same base name, such as `myproj.iwp` and `myproj.iwo`.

### Tip



If you inadvertently delete the options file associated with your project, you can recover your project by recreating an .IWO file with the same name that contains a simple \*.

To create a WorkFrame project file type, follow these steps:

### Under Windows 95

1. From the desktop, double-click on My Computer icon
2. Choose **View**→**Options...**
3. Switch to the *File Types* notebook page
4. The *IWP* file type is already listed, select it and click on **Edit...**
5. In the *Description of type* field, enter: **WorkFrame Project**
6. Select the open action, and click on **Edit...**
7. In the *Application Used to Perform Action* field, enter: `D:\IBM-CPPW\BIN\Iwfwf35.exe`
8. Click on **OK** and close all the dialog windows that remain. Changes are applied immediately.

### Under Windows NT version 3.51

1. Open the File Manager
2. Choose **File**→**Associate...**
3. Choose *New Type...*
4. In the File Type entry field, enter: **WorkFrame Project**

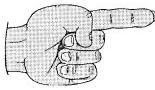
5. From the **Action** combination box, select **Open**, and enter **D:\IBM-CPPW\BIN\Iwfwf35.exe** in the *Command* entry field.
6. In the *New Extension* entry field enter: **iwp**, and click on the **Add** push button.
7. Click on **OK** to confirm the type creation.

## Creating and Customizing the Dacslib Project

The Dacslib project is used to manage data created from Data Access Builder. The strategy is to map the different DB2 tables used in a subsystem such as Property to C++ classes and group all of those classes into a single library.

The Dacslib project definition is based on the VisualAge for C++ default project. The VisualAge for C++ default project actions and types are defined in the VACPP.IWS file, located in the D:\IBMCPP\MAINPRJ directory. Modifications to the VACPP.IWS file affect any project you create from Project Smarts, and can hurt the way MakeMake works. See “Modifying the WorkFrame Configuration File” on page 123 for more information about the contents of this file.

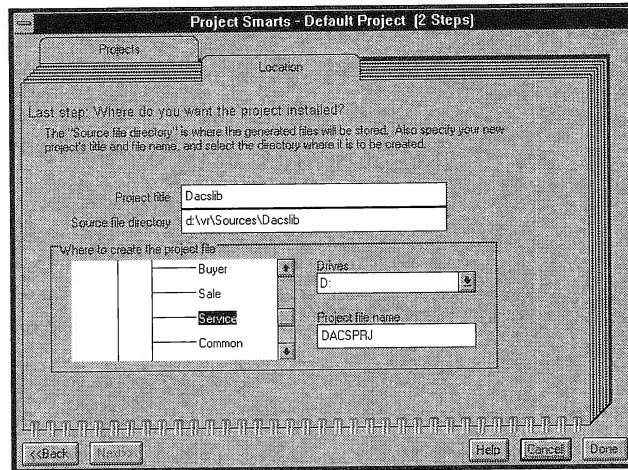
### Read This



With Project Smarts, you can create a sample Data Access Builder project that contains files to use as a startup for your development. We do not use this sample project as we already have files generated from the Property subsystem tables and views.

## Creating the Dacslib Project

To create the Dacslib project, start WorkFrame by clicking on the **WorkFrame IDE** icon, then choose **Create New Project**. Select **Default Project** and click on **Next>>**. You now have to fill in the data as shown on Figure 43.



**Figure 43.** Location Settings for the Dacslib Project

The following information must be supplied to create the Dacslib project:

<b>Project title</b>	Dacslib
<b>Source file directory</b>	D:\VR\Sources\Dacslib  This directory contains the files that the Dacslib project manages. Note that you can specify only one directory. If your project is supposed to manage several directories, you must change its settings once it is created.
<b>Where to create the project file</b>	D:\VR\Sources\Service  The Dacslib project is contained within the Service project (see Figure 41). For example, ProjectA is contained by ProjectB if it is created in one of the directories managed by ProjectB. Thus, you create the Dacslib project inside the Service project working directory.
<b>Project file name</b>	DACSPRJ  Two files are created for this project: DACSPRJ.IWP and DACSPRJ.IWO.

Click on **Done** to create the project. WorkFrame automatically opens the newly created Dacslib project. You can now customize it.

## Customizing the Dacslib Project

You must modify the Dacslib project as follows:

- ☐ Change the Dacslib project settings: specify the project target and make file name and set values for the project's environment variables
- ☐ Change the DB2 Precompile action flags
- ☐ Change Compile action flags
- ☐ Change Link action flags
- ☐ Set the Build facility options.

### *Changing the Dacslib Project Settings*

To modify the project target settings, select *View→Settings→Target* from the project toolbar and then specify the following information:

<b>Target Name</b>	..\..\Bin\vrtdacs.dll
<b>Makefile</b>	vrtdacs.mak

You must now specify some environment variables to ensure that the Make action works correctly. Environment variables are local to the project, and known **only** if you start the Make action from the Dacslib project. In other words, compilation will fail if it is started from a command window.

You must specify environment variables in the following situations:

- ☐ Your project is using header files managed by a different project, or stored in a directory that is not listed in the system INCLUDE environment variable.
- ☐ Your project requires libraries at the link step that are not pointed by any directory listed in the system LIB environment variable.
- ☐ Your project's target uses DLLs at runtime that are stored in a directory that is not listed in the system PATH environment variable.
- ☐ Your project is using a help file that is not stored in any directory listed in the system HELP environment variable.

For the Dacslib project, you have to specify a new value for the LIB environment variable so that the export (.exp) and library (.lib) files created by the IBM librarian manager (ilib) in the Bin directory are found by the linker. To do this, select **View→Settings→Environment Variables** from the project toolbar and type the following line:

```
LIB=..\..\Bin;%LIB%
```

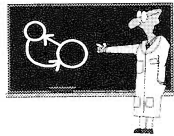
By specifying a relative path, you ensure the portability of the development hierarchy.

## Changing DB2 Precompile Action Flags

Because Data Access Builder uses DB2 Embedded SQL in our application, the DB2 Precompile action must be configured to automatically generate bind files (.bnd files) as well as the database packages. In other words, the BIND and PACKAGE flags must be turned on:

1. Select **Options**→**DB2 Precompile** from the Dacslib project menu bar.
2. On the Common notebook page, click in the BIND and PACKAGE check boxes to turn those options on. You do not have to specify any output file names.
3. Click on **OK** to commit the changes.

### Technical Information



Data Access Builder generates some information relative to the database used, such as its name. If you want to ignore the information generated, and prompt for the database name, user ID and password, then you must click in the two following check boxes:

- Ignore generated database name
- Prompt for Database, Userid, or Authentication

The Dacslib project must now be configured so that the compilation and linkage flags are correctly set to generate a DLL from the Data Access Builder data.

## Changing Flags for Compiler Action

Choose **Options**→**Compile** from the Options menu bar item. In the C++ Compiler options dialog you can specify the following options:

- ☐ Target: **DLL** (on the Processing notebook page)
- ☐ Object: **..\..\Objects\%n** (on the File notebook page)

This option forces the creation of all object files into the `..\..\Objects\%n` path. Using this option is equivalent to using the `/Fo` compiler option.

Be sure to specify only a relative path in this field. Specifying an absolute path causes the Make action to fail.

The `%n` substitution variable represents the name of the current file being processed by the make file, without an extension and path. For example, if NMake processes the `..\sources\Property\property.cpp` file then `%n` contains `property`.

- ☐ Template: **..\..\Objects\DACSLIB** (on the File notebook page)

This option forces the compiler to generate all template-related files into the **..\..\Objects\DACSLIB** directory. This option is equivalent to using the **/Ft** compiler option.

#### Warning



You must apply the following changes to the **VACPP.IWS** file located in the **D:\IBMCPW\MAINPRJ** directory so that MakeMake generates a correct Makefile from the previous settings:

- ☐ Backup the **VACPP.IWS** file to **VACPP.IWS.BAK**, then edit it.
- ☐ Locate the *FileTools::Make Exp File* entry in that file
- ☐ In the tool definition, replace **-target=%q%t.def** by **-target=%t.def**.

- ☐ Library selection: **Multithread** (on the Object page)

You must specify this option (equivalent to **/Gm**) if you are using the IBM Open Class library.

- ☐ Library linkage: **Dynamic** (on the Object page)

You must specify this option (equivalent to **/Gd**) if you want to create a dynamically linked library.

- ☐ Target Processor: Specify whatever is suitable for your machine. Specify the **Blend** value if you want your code to run on any processor, from the 80386 to the Pentium Pro processor.

Click the **OK** push-button to close the window and save the changes.

### Changing Flags for Link Action

Open the linking options by selecting **Options→Link** and specify the following options:

- ☐ Application Type: **None** (on the Generation notebook page)
- ☐ Run-file: **Dynamic Link Library** (On the Definition/RunTime notebook page)
- ☐ C++ Templates Support: You have to specify that the generated library is using templates by selecting **Templates Used** in the templates page (On the Templates notebook page)

Click the **OK** push-button to close the window and save the changes.

## Setting the Build Facility Options

The Build facility creates a make file for the project and tries to build the project from it. To correctly generate the make file, you have to set up the list of actions that the MakeMake facility uses. To configure the Build action, select **Options**→**Build Normal** from the Dacslib project menu bar, and then select the following actions:

- ☐ Compile
- ☐ DB2 Precompile
- ☐ Link
- ☐ Make exp and lib

By specifying the Make exp and lib option, you trigger a call to the IBM Librarian Manager (ilib). Ilib creates an export file (.exp) from the list of object files that you want to store in the library, as well as the corresponding library file (.lib) that other applications need to link with your library. Both files are created in the Bin directory.

# Creating and Customizing the Visual Realty Projects

The Dacslib project is now ready for use. The next step is to create the main project, Visual Realty, and the various subsystem projects.

## Creating the Visual Realty Main and Subsystem Projects

The various projects for each subsystem are created with Project Smarts from the default VisualAge for C++ project. To create the Property project:

1. Double-click on the **WorkFrame IDE** icon.
2. Click on **Create new project**.
3. Select **Default Project** application from the available project list.
4. Click on **Next>>**.
5. In the dialog window, enter the following data:

<b>Project title</b>	Property
<b>Source file directory</b>	D:\VR\Sources\Property
<b>Where to create the project files</b>	D:\VR\Sources\Main
<b>Project file name</b>	PROPERTY

Repeat Steps 3, 4, and 5 to create the Common, Service, and Visual Realty projects with the following data:



Project Title	Source File Directory	Where to create the Project File	Project Name
Common	D:\VR\Sources\Common	D:\VR\Sources\Service	COMMON
Service	D:\VR\Sources\Service	D:\VR\Sources\Property	SERVICE
Visual Realty	D:\VR\Sources>Main	D:\VR\Sources	VRMAIN

You must now change the settings of each project to set its target name and make file name. For each project, open the project settings, switch to the Target notebook page, and change the target name and make file name as follows:

Project	Target Name	Make File
Property	..\..\Bin\vrprop.dll	vrprop.mak
Service	..\..\Bin\vrserv.dll	vrserv.mak
Common	..\..\Bin\vrcomm.dll	vrcomm.mak
Visual Realty	..\..\Bin\vrmain.exe	vrmain.mak

## Creating the Help Project

The Help project is aimed at managing the context-sensitive help files of the application. It is defined as a subproject of the Visual Realty project. You create it from Project Smarts.

To create the Help project:

1. Double-click on the **WorkFrame IDE** icon.
2. Click the **Create new project** push-button.
3. Select **IPF Document or Help** from the available project list.
4. Click on **Next>>**.
5. In the dialog window, enter the following data:
 

<b>Project title</b>	Help
<b>Source file directory</b>	D:\VR\Help
<b>Where to create the project files</b>	D:\VR\Sources>Main
<b>Project file name</b>	VRHELP
6. Click on **Next>>**
7. In the Document type group box, choose the **Help** radio button.

8. Click on **Done** to create the project.

**Tip**



When you use Project Smarts to create a project that is not the default project, some sample files are created for you. You can delete those files if they are not of any use for your project. In our sample application, you must always delete them.

You are now ready to customize the Visual Realty projects.

## Customizing the Visual Realty Main and Subsystem Projects

To correctly compile the Service, Property, and Visual Realty projects, you must:

- ☐ Set the values of the LIB and INCLUDE environment variables: Those variables are used by the compiler or the linker to find header files and libraries.
- ☐ Set the Visual Realty subsystems projects to generate a DLL
- ☐ Set the Visual Realty main project to generate an executable
- ☐ Set the compile and link flags for all projects
- ☐ Set the Build facility options
- ☐ Set various variables and flags to activate the trace facility in the User Interface Class Library.

### Modifying the LIB and INCLUDE Environment Variables

In the Visual Realty application, common parts are grouped in the Common project. Both the Service and Property subsystems use those common parts. You therefore must update the LIB and INCLUDE variables to include the path where common includes and libraries are located for the Property and Service projects. The Property subsystem also uses the Service and Dacslib subsystems. The LIB and INCLUDE variables must be set up accordingly. Note that the Common project does not depend any upon any other project. Therefore, it does not require such a setting.

**Warning**



The new value of an environment variable is known only *within the scope* of the project. If your make file requires the newly defined INCLUDE variable to correctly build the application, you must start the make action from the Work-Frame project. If you start the make action from a command window, the INCLUDE value is as defined in the system variables (Windows NT) or your AUTOEXEC.BAT file (Windows 95).

To define the LIB and INCLUDE variables for the Property project:

1. Select **View**→**Settings**→**Environment Variables**.
2. Enter the following information:

LIB=..\..\Bin;%LIB%

INCLUDE=..\Common;..\Service;..\Dacslib;%INCLUDE%

Repeat Steps 1 and 2 to define the LIB, INCLUDE variables for the Visual Realty and Service projects using the following table. Note that you must also define the HELP variable for the Visual Realty project to ensure that the help files are found when you start the application.

Project Name	Variable	Line To Add
Visual Realty	LIB	..\..\Bin;%LIB%
Visual Realty	INCLUDE	..\Common;..\Service;..\Property;..\Dacslib; %INCLUDE%
Visual Realty	HELP	..\..\Help;%HELP%
Service	LIB	..\..\Bin;%BIN%
Service	INCLUDE	..\Common;%INCLUDE%

### ***Setting Up the Subsystem Projects to Generate a DLL***

Each Visual Realty subsystem is compiled and linked to generate a DLL. Therefore, you must change the compile and linking flags for each subsystem.

For each subsystem project (Common, Service, and Property), open the project and choose **Options**→**Compile**. Switch to the Processing notebook page and do the following:

1. In the Processing Step group box, select the **Perform compile only** radio button.
2. In the Target group box, select the **DLL** radio button.
3. Click on **OK** to commit the changes.

Choose **Options**→**Link**.

1. In the Application Type group box, select **None**.
2. Switch to the Definition/Run-file notebook page.
3. In the Run-File group box, select the **Dynamic link library** radio button.
4. Click on **OK** to commit the changes.

## Setting Up the Visual Reality Project to Generate an EXE

Open the Visual Reality project, and choose **Options→Compile**.

1. Switch to the Processing notebook page.
2. In the Processing step group box, select the **Perform compile only** radio button
3. In the Target group box, select the **EXE** radio button.
4. Click on **OK** to commit the changes

Choose **Options→Link**.

1. In the Application Type group-box, select the PM type. This ensures that the application is linked correctly to run in a window. This is equivalent to using the /pmttype:pm option on a command line.

**Note:** You must not use the /pmttype linker option when you generate DLLs.

2. Switch to the Definition/Run-file notebook page.
3. In the Run-file group box, select the **Executable** radio button.
4. Click on **OK** to commit the changes.

## Setting Up the Compile Action Options

For each subproject, you need to specify where the objects files, as well as the template-generated files, must be created. To do so:

1. Select **Options→Compile** from the project toolbar
2. In the File notebook page, enter the following values:
  - Object: `..\..\Objects\%n`
  - Template: `..\..\Objects\VRXXXX` where XXXX uniquely identifies the directory where template files are to be put.
3. Click on **OK** to commit the changes.

### Warning



You must apply the following changes to the VACPP.IWS file located in the D:\IBMCPWP\MAINPRJ directory so that MakeMake generates a correct Makefile from the previous settings:

- ☐ Backup the VACPP.IWS file to VACPPIWS.BAK, then edit it.
- ☐ Locate the *FileTools::Make Exp File* entry in that file
- ☐ In the tool definition, replace **-target=%q%t.def** by **-target=%t.def**.

### ***Setting Up the Linking Action Options***

Each project requires some libraries to build correctly. For example, the Property project uses the Common project and Service project libraries. Therefore, you must specify that those libraries are used at the link step. To do this, select **Options→Link** for each of the following projects, switch to the File Names notebook page, and specify the following library information:

Project Name	Add Libraries
Property	vrcomm.lib vrserv.lib
Visual Realty	vrcomm.lib vrserv.lib vrprop.lib

Switch to the Templates page. Ensure that the **Templates used** check box is selected for every project.

### ***Setting Up the Build Facility Options***

To correctly generate the make file, you have to set up the list of actions that the MakeMake facility uses. To configure the Build action, open each of the Service, Common, and Property projects, select **Options→Build Normal**, and select the following actions:

- ☐ Compile
- ☐ Link
- ☐ Make Exp and Lib

For the Visual Realty project, select the following actions:

- ☐ Compile
- ☐ Link
- ☐ Resource Compile
- ☐ Resource Precompile

Repeat these two steps for the **Options→Rebuild All** option.

### ***Setting Up a Project for Trace Support***

The User Interface Class library provides a facility to trace any methods, or show any exception that is thrown. This trace facility is particularly useful if you want to debug code that has been generated by Visual Builder. As a matter of fact, Visual Builder generates some trace code for each connection that you have created. By activating the trace facility, you can easily follow the flow of logic in your code, that is the order in which the connections are fired. To activate the trace facility in the User Interface class library, follow these steps:

1. Compile your code with the `IC_TRACE_DEVELOP` preprocessor macro. You can set this macro from the Build Smarts facility:

- a. Open Build Smarts and select the **Development** check button
  - b. In the *Define* entry field, enter IC\_TRACE\_DEVELOP.
2. Set the ICLUI\_TRACE and ICLUI\_TRACETO environment variables to enable the trace functions. The ICLUI\_TRACE variable enables or disables the trace according to its value (ON/OFF), and the ICLUI\_TRACETO variable enables you to redirect the trace output to a file or a standard output. Redirect the output to STDOUT so that you can see it from the project monitor window.

To set the ICLUI\_TRACE and ICLUI\_TRACETO variables you can either:

- Windows 95: Add the following statements to your AUTOEXEC.BAT file:

```
SET ICLUI_TRACE=ON
SET ICLUI_TRACETO=STDOUT
```

- Windows NT: Open the System settings from the Control Panel and enter the following data:

<b>Variable</b>	ICLUI_TRACE
<b>Value</b>	ON

Click on **Set** to create the variable in your user environment. Proceed the same way to add the ICLUI\_TRACETO variable with the value STDOUT.

or

- Add the variables to the project environment

Open the project and select **View**→**Settings**→**Environment Variables**. Enter the following information to define the ICLUI\_TRACE and ICLUI\_TRACETO variables.

```
ICLUI_TRACE=ON
ICLUI_TRACETO=STDOUT
```

If you want to redirect the output to a file, you have to define the following environment variables:

```
ICLUI_TRACE = ON
ICLUI_TRACETO = FILE
ICLUI_TRACEFILE = filename
```

### ***Setting Up a Project for Visual Builder***

After you have used Visual Builder for a while, you may find it annoying to have to manually load the necessary VBB files for your application. Visual Builder provides a facility for automatically loading the correct VBBs when you start the tool. All you have to do is create a

VBLOAD.DAT file, using your favorite editor. A VBLOAD.DAT file is a flat file in which you list the VBB files you want to load at startup. The VBLOAD.DAT file for the Property subsystem looks like this:

```
D:\IBMCPW\IVB\VBDAX.VBB
D:\IBMCPW\IVB\VBMM.VBB
D:\IBMCPW\IVB\VBSAMPLE.VBB
D:\VR\Sources\Property\VRPROP.VBB
D:\VR\Sources\Common\VRCOMM.VBB
D:\VR\Sources\Service\VRSERV.VBB
```

You must specify the full path name for each VBB. Note that you do not have to specify the VBBase.VBB file because Visual Builder always loads it at startup.

For each Visual Realty project, create a VBLOAD.DAT file as follows:

Project Name	VBLOAD.DAT File Contents
Service	D:\IBMCPW\IVB\VBDAX.VBB D:\IBMCPW\IVB\VBSAMPLE.VBB D:\VR\Sources\Common\VRCOMM.VBB D:\VR\Sources\Common\KBDHDR.VBB D:\VR\Sources\Service\VRSERV.VBB
Common	D:\IBMCPW\IVB\VBSAMPLE.VBB D:\VR\Sources\Common\VRCOMM.VBB
Visual Realty	D:\IBMCPW\IVB\VBDAX.VBB D:\IBMCPW\IVB\VBSAMPLE.VBB D:\VR\Sources\Common\VRCOMM.VBB D:\VR\Sources\Common\KBDHDR.VBB D:\VR\Sources\Service\VRSERV.VBB D:\VR\Sources\Property\VRPROP.VBB D:\VR\Sources\Main\VRMAIN.VBB

## Naming Conventions

We used the following name conventions throughout the development of the Visual Realty application (see Appendix D, “Class Dictionary,” on page 521):

1. All file names are built according to the FAT format (eight letters for the file name, three letters for the file extension).
2. All file names are built according to the VRSXXXXX format, where:

- VR is an invariant for Visual Realty
- S is the subsystem name
  - B, for the Buyer subsystem
  - P, for the Property subsystem
  - T, for the Sales Transaction subsystem
  - C, for the Common subsystem
  - S, for the Service subsystem
  - XXXXX identifies the file contents.

## Run-Time Considerations

For the application to run, the necessary DLLs must be located in one of the directories listed in the PATH variable. We therefore created a D:\VR\Bin directory where all of the Visual Realty application DLLs as well as the application executable are created. This directory must be added to the PATH variable in the AUTOEXEC.BAT file (for Windows 95) or to your system variables (for Windows NT).

## Modifying the WorkFrame Configuration File

### Disclaimer



You may change the WorkFrame solution configuration file. However:

- ☐ IBM will not be supporting the format of the WorkFrame configuration file in future versions of VisualAge for C++. Thus, any changes made to the configuration file will not be migrated to a future version.
- ☐ Errors in the configuration file can prevent WorkFrame from operating correctly. Be sure to back up the original configuration file before changing it.

The WorkFrame configuration file contains the configuration for types, tools, options, and GUI setup preferences used by the WorkFrame component. This file cannot be dynamically changed in the VisualAge for C++ product, but an in-depth knowledge of the file contents may allow you to statically customize the entries in the file that are configurable.



The WorkFrame configuration file is located in the D:\IBMCPW\MAIN-PRJ directory and is called VACPP.IWS. Before attempting any changes to this file, you should back it up, as any incorrect changes might cause a failure in operation of the WorkFrame component or other VisualAge for C++ tool.

Types and actions definitions stored in the WorkFrame configuration file are taken into account:

- ☐ When you create a project by using Project Smarts
- ☐ When you open an existing project

### ***WorkFrame Configuration File Contents***

All entries in the configuration file have the same format. Each entry is composed of an entry type, an entry name, one or several entry keys, and one or several key values. For example, the first type definition in the file describes the “Project File” type:

```
-Types::Project File
+mask=*.iwp
```

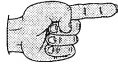
In the above example, *Types* represents an entry type, *Project File* is the entry name (always separated from the entry type with a “::” sign), *mask* is the entry key (always prefixed with a “+” sign and followed by an “=” sign), and *\*.iwp* is the key value. Here *\*.iwp* represents a file mask.

In addition to file mask definitions, the WorkFrame configuration file uses the same syntax to define project and file scoped actions, GUI icons, options menus, the project toolbar menu, the project menu, and the project menu shown in other VisualAge for C++ tools.

## **Customization Possibilities**

By editing the VACPP.IWS file, you may configure the following:

- ☐ Add or modify WorkFrame types
- ☐ Add or modify WorkFrame actions
- ☐ Associate an action with an Icon
- ☐ Change the WorkFrame project toolbar
- ☐ Change the WorkFrame project menu
- ☐ Change the VisualAge for C++ tools project men

**Read This**

The project options menu is **not** configurable, and must not be changed.

## Types Customization

Types are used to categorize project parts. An example of a type is “VBFile” which describes file whose names match the \*.vbb file mask. Types can be used in two different ways:

- ☐ To determine the actions that apply to one or several selected project parts. For example, the “Visual” action applies to the “VBFile” type.
- ☐ To determine the target that an action can produce, for example the “C++Compiler” action can produce “ObjectFiles.”

You may create new types or modify existing ones. For example, if you want to create a new type for Java files, you must create the following entries in the Types section of the VACPP.IWS file:

```
-Types::JavaSource          -Types::JavaOut
+mask=*.java;*.jav         +mask=*.class
```

You must be cautious if you modify existing types: Removing file masks from type definitions also dissociates the files corresponding to that file mask from the action that is associated to the type. For example, if you remove the \*.cpp file mask from the C++Source type definition, then you will not be able to invoke the C++ compiler on a C++ source file that matches the \*.cpp extension.

## Actions Customization

An action can be defined as:

- ☐ File-scoped

File-scoped actions apply to specific project parts, and can be invoked from those parts. To run a file-scope action, you must select one or several project parts, then invoke the action on the selected parts. Note that only file-scoped actions are eligible to MakeMake, and thus can be included in a Makefile.

To create a file-scoped action, you must use the FileTools entry type.

### ❑ Project-scoped

Project-scoped actions apply to a project as a whole, rather than to a project part. Project information can be passed to a project-scoped action, such as the name of the makefile or the project's target. Build Normal or Run are examples of project-scoped actions. Project-scoped actions cannot be included in a Makefile.

To create a project-scoped action, you must use the `ProjectTools` entry type.

An action is described in terms of an executable, a name, an input type, and optionally an output type. Actions may only vary in the parameters supplied to run them. For example, the LPEX Editor is used in two actions: the LPEX Editor action allows you to edit a file and modify it while the LPEX Browser action allows you to view a file in read-only mode. Those actions differ only by the parameters supplied when they are invoked on a file..

#### Tip



The ordering of file-scoped actions definition in the `VACPP.IWS` file is important. The nearer the action to the top of the list, the higher the priority. When you double-click on a file, then the highest priority action which is valid for the file type is called.

If you were to follow the Java example, and set up the Java run-time environment, you would add the two following file-scoped actions:

```
+FileTools::JavaCompile
+title=Java Compile
+program=javac.exe
+accelerator=J
+helpCmd=iview javabook %TOPIC%
+topic=Compiling Java Applications
+input=JavaSource
+output=JavaOut
+parm=%f

+FileTools::JavaRun
+title=Java Run
+program=java.exe
+accelerator=A
+helpCmd=iview javabook %TOPIC%
+topic=Running Java Applications
+input=JavaOut
+parm=%f
```

**Note:** the accelerator key has to be unique amongst all tools.

The *parm* key has predefined replacement arguments (such as %f) with the following meanings:

Replacement Argument	Meaning
%a fill %z	All selected project parts, each separated by the “fill” string. The default “fill string” is a space.
%d	The project’s working directory.
%e	The extension of the first selected project part.
%f	The fully qualified name of the first selected project part.
%m	The project’s makefile name.
%n	The name portion (less path and extension) of the first selected project part.
%o	The project’s target.
%p	The project’s filename.
%q	The qualified portion of the path of the first selected project part.
%r	The project’s run options, as defined in the project’s setting notebook.
%t	The fully qualified name (without an extension) of the project’s target.
%TOPIC%	The value of the topic entry key. This entry specifies the help topic to be displayed when the user requests help on the action.

If the first selected file is: D:\VR\Sources\vrmain.cpp then:

- ☐ %f is D:\VR\Sources\vrmain.cpp
- ☐ %q is D:\VR\Sources
- ☐ %n is vrmain
- ☐ %e is .cpp

You may also use the following entry keys to define an action:

Entry Key	Values	Meaning
setenv	yes no	Specifies whether WorkFrame sets the environment variables before running a tool. The default value is no.
runmode	mon def	Specifies whether a tool should be run in monitored mode (output/input made through project monitor window) or in default mode. The default mode is to run the tool in a window.

Entry Key	Values	Meaning
optionsDll		<p>The action support DLL allows you for example, to display a GUI to gather the values for action options such as compiler or linking flags. You must use this entry key if you want to define an action related to the compiler or the linker.</p> <p><b>Note:</b> you cannot define your own DLL, as the WorkFrame API is not provided with the product.</p>
entryPoint		<p>Since an action support DLL can provide support for several actions, you must also specify an entry point into that DLL.</p>

### Icons Customization

The WorkFrame GUI contains files which are associated with icons through their file type. This is configured by using “Icons::” entries in the WorkFrame configuration file. To change or add icons for file types, you need to build a DLL containing only icons and replace the default DLL defined in the VACPP.IWS file (IWFWIC35.DLL) with this new library. The original icon files are not supplied. Thus, you must supply an icon for each ID already defined in the VACPP.IWS file. To define the default DLL for icons, you must replace:

```
-IconsControl::Data
+resourceDLL=IWFWIC35
+defaultIcon=1001
```

with:

```
-IconsControl::Data
+resourceDLL=myDLLName
+defaultIcon=XX01
```

**Note:** Only the IconsControl entry type is required. The icon definition for a type is optional. If an icon ID has not been defined for a type, then the default icon ID is used.

If you have specified your own DLL for the IconsControl entry, then you must replace the resource ID of each icon already defined by the corresponding resource ID in your DLL. For example:

```
-Icons::Runnable
+mask=Runnable
+id=1002
```

becomes

```
-Icons::Runnable
+mask=Runnable
+id=XX02
```

In the same way, you can define an icon for new types such as JavaSource and JavaOut as follows:

```
-Icons::JavaSrc          -Icons::JavaCode
+mask::JavaSource        +mask::JavaOut
+id=XX06                 +id=XX07
```

**Note:** XX in XX01 can be any number.

### ***Toolbar Customization***

The toolbar of a WorkFrame project may be configured in a similar way to icons. A DLL must be built with bitmaps associated with resource ids for all the previously defined tools that you want to show on the toolbar. For example, if you want to add the JavaCompile and JavaRun actions to the toolbar, you would define the following entries:

```
-Toolbar::Item12          -Toolbar::Item13
+tool::JavaCompile        +tool::JavaRun
+scope=file               +scope=file
+id=XX12                  +id=XX13
+iconTitle=JavaC          +iconTitle=JavaR
```

### ***Resource DLL Creation***

You can create a resource DLL as follows:

- Using Project Smarts, create a resource library project.
- Create an empty.c file with contents: `void empty() {}`
- Customize the resource file that has been generated by Project Smarts according to the following format:

```
1 icon xxx.ico
2 icon yyy.ico
...
1000 bitmap xxx.bmp
1001 bitmap yyy.bmp
```

You can either create this file using your favorite editor, or use the Resource Workshop. Note that the Resource Workshop provides an icon editor as well as a bitmap editor.

- Select **Project->Build Normal**.

A Resource DLL is created as your project's target. You can use this DLL to load your resources in your application dynamically (refer to `ADeleteDialogView` on page 183 for more information on how to use a resource DLL).

### ***Project Menu Customization***

You can customize the WorkFrame project menu by adding or removing "ProjectMenu" entries in the WorkFrame configuration file. This menu is displayed when you select *Project* from a project menu bar. For example, the *Build Normal* menu item is declared as follows:

```
-ProjectMenu::Item1  
+tool=Build Normal
```

The toolbar is built based on the sequence of entries in the `VACPP.IWS` file, and regardless of the item number that you have declared.

### ***VisualAge for C++ Tools Project Menu Customization***

The project menu of other VisualAge for C++ tools is specified in the WorkFrame solution file through "Tribble Menu" entries. This menu is shown when you start a tool from within a WorkFrame project. This menu always contains all entries of the WorkFrame project menu described above, plus the "Tribble Menu" entries in the WorkFrame solution file.

# 6

## Mapping Relational Tables Using Data Access Builder

*Well begun is half done.*

-Proverb

The first parts you create, you do not develop yourself. Instead you use Data Access Builder to generate them. We discuss the features of Data Access Builder in Chapter 2, “Getting Started in a VisualAge for C++ Environment,” on page 19. In this chapter we show you how to use Data Access Builder to map the tables of the property subsystem to parts. You learn about the purpose of the generated files and classes. Eventually, you build a simple application that displays the data in one table in a list box.



We assume in this chapter that you have installed DB2 Single-User and run Windows NT. When you install the application, the database is created automatically. See Chapter 11, “More about Data Access Builder...,” on page 373 if you need details to set up DB2 Client Access Enabler for Windows 95 or use dbf files as a data source together with an appropriate ODBC driver.

## Defining the Tables and Views

In the design object model (Figure 40 on page 100), *property* is described by the following classes and their relationships:

<b>Property</b>	Holds the general property information: <ul style="list-style-type: none"><li>Property registration identifier</li><li>Size</li><li>Number of stories</li><li>Number of bathrooms</li><li>Number of bedrooms</li><li>Cooling type</li><li>Heating type</li><li>Description</li></ul>
<b>Address</b>	Holds the location information for the property: <ul style="list-style-type: none"><li>Street</li><li>City</li><li>Area</li><li>State</li><li>Zip code</li></ul>
<b>MarketingInfo</b>	Holds the marketing information for the property: <ul style="list-style-type: none"><li>Price</li><li>Days on the market (elapsed time between the last update and the creation date)</li><li>Commission rate (agent commission rate)</li><li>Down payment rate (buyer payment rate)</li></ul>
<b>PropertyLog</b>	Holds the logging information: <ul style="list-style-type: none"><li>Creation time stamp</li><li>Last update</li><li>Status (available, pending, or sold)</li></ul>
<b>MultiDoc</b>	Holds the multimedia information for the property: <ul style="list-style-type: none"><li>Type (bitmap or video; only the video features are implemented in the sample application)</li><li>File name</li></ul>

To store the property information, we define the following five tables (see Appendix C, “Database Definition,” on page 515 for details):

- ☐ PROPERTY
- ☐ PROPERTY\_ADDRESS
- ☐ MARKETING\_INFO
- ☐ PROPERTY\_LOG
- ☐ MULTI\_DOC

The relationships between Property and the other classes are represented by a foreign key in each table. In addition, we create two views:

- ☐ PROP\_AD\_LOG is built by joining the PROPERTY, PROPERTY\_ADDRESS, MARKETING\_INFO and PROPERTY\_LOG tables on PROPERTY\_ID and ADDRESS\_ID. It is used to simultaneously display, in a container, the information from the four tables (see “Using a Container” on page 192).
- ☐ LIST\_AREA is built by joining the PROPERTY\_ADDRESS and PROPERTY\_LOG tables and contains, in alphabetic order, distinct areas in which available properties are located.

## Mapping Tables to Parts

You can start Data Access Builder in three ways:

- ☐ Click on the appropriate icon in the IBM VisualAge for C++ for Windows folder.
- ☐ Start the product from a command prompt.
- ☐ Start the product from Workframe.

With the first method, the source files are generated in the directory that is specified in the program properties. By default, the files are produced in the D:\IBMCPW\WORKING directory (assuming you have installed VisualAge for C++ on your D drive in the default directory). If you want to generate your code in the D:\REAL directory (assuming you have created a \REAL directory to hold all files related to your application), open the properties dialog of Data Access Builder and set the Start in directory to D:\REAL.

With the second method, the code is generated in the directory from which you start Data Access Builder. Therefore, if you want to generate your code in D:\REAL, simply change to the D:\REAL directory and type the `idata` command to start Data Access Builder.

Use the third method to start Data Access Builder. In Chapter 5, “Setting Up the Development Environment,” on page 105, you defined a Dacslib project for the Property subsystem. From Dacslib you can start Data Access Builder in four ways:

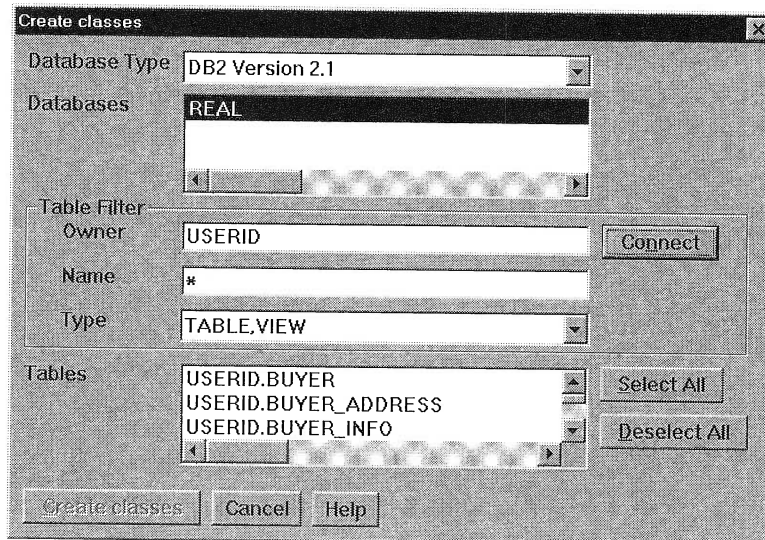
- ☐ Select the **Database** option from the Project pull-down menu.

- ☐ Select the **Database** option from the project's pop-up menu.
- ☐ Use the accelerator keys: **Ctrl+Shift+A**.
- ☐ Double-click on a Data Access Builder session file (extension DAX).

Data Access Builder generates the files in the directory that you specified on the Directories page of the Dacslib project settings notebook, that is, D:\VR\SOURCES\DACSLIB.

Open the DACS library project and start Data Access Builder. In the startup window you can either create new classes from relational tables or open a Data Access Builder file and resume the work of a saved session

To create a mapping from scratch, click on the **Create Classes...** push button. The database type defines the data access method (see Figure 44). Select DB2 Version 2.1 as the database type. Data Access Builder accesses your database directory and presents a list of all database names cataloged on your machine. When you select the database type ODBC Data Sources, a list of registered ODBC data sources is displayed. From the list, you can select the database you want to work with.

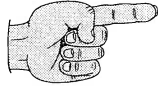


**Figure 44.** Data Access Builder Create Classes Window

Select the **REAL** database, the database owner (usually the database is created by the local administrator **USERID**), the **TABLE, VIEW** type, and click on **Connect**.

When you click on **Connect**, Data Access Builder tries to connect to the selected database. If DB2 Database Manager is not already running, it is started. If you are not logged on, you are asked to give the user ID and password of the database creator (in our case, the user ID is **userid**, and the password is **password**).

**Read this**

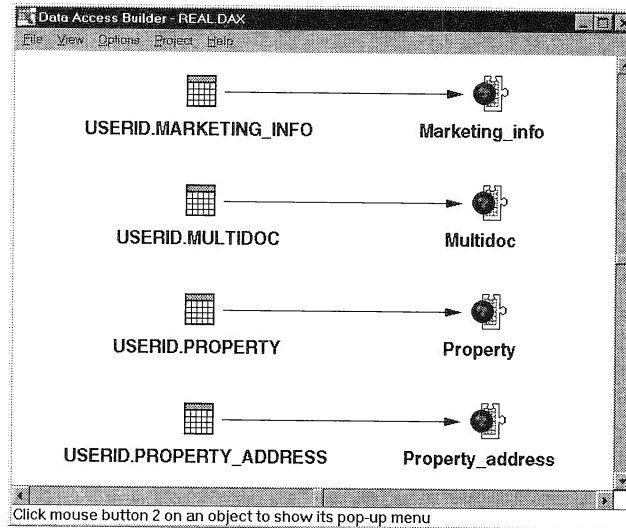


DB2 passes userid and password to the Windows NT authentication system. Windows NT validates user ID case-insensitive and the password case-sensitive. DB2 will always show the user ID in uppercase. If you specify the owner table filter in the Create classes dialog, you should type the userid in uppercase.

Once the database connection is established, Data Access Builder lists the tables and views, prefixed with USERID (Figure 44). Click on each table or view you want to map to a part. In our case, select only the tables and views related to the Property subsystem:

- ☐ USERID.LIST\_AREA
- ☐ USERID.MARKETING\_INFO
- ☐ USERID.MULTI\_DOC
- ☐ USERID.PROPERTY
- ☐ USERID.PROPERTY\_ADDRESS
- ☐ USERID.PROPERTY\_LOG
- ☐ USERID.PROP\_AD\_LOG

Click on **Create classes**. Data Access Builder prompts you for the class options. By default, Data Access Builder generates C++ Visual Builder parts and accesses the database with the embedded SQL method. Click on **OK** to confirm these options and to get to the primary window, which shows the mapping of each table on the free-form surface (Figure 45).



**Figure 45.** Data Access Builder Main Window

To access the settings window of each mapped class, click on it with the right mouse button and select **Open settings** from the pop-up menu or, as a shortcut, double-click with the left mouse button.

Before you generate the code for each part, you can check the associated file names in the Names page. The files generated for each part depend on your class options and whether you chose to generate a make file. In our case the following files are generated:

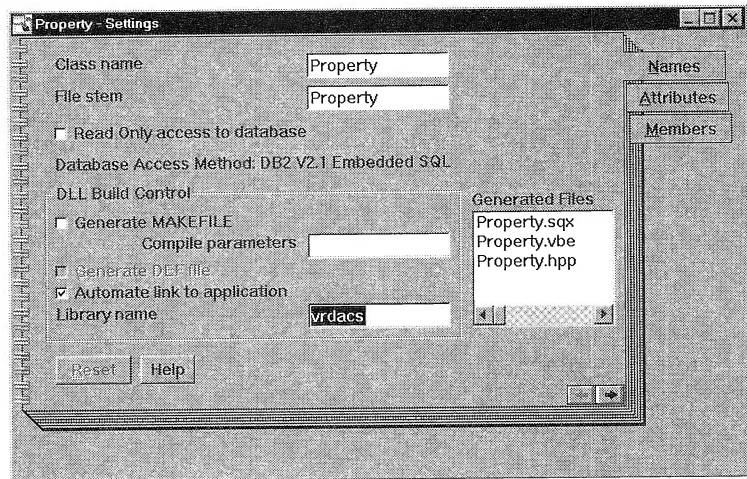
- ☐ \*.hpp, header for the parts
- ☐ \*.sqx, part code with embedded SQL
- ☐ \*.vbe, import file for Visual Builder

You use a short file stem length so that all development tools can access the source code. You can modify the file stem with the following names:

- ☐ ListArea for LIST\_AREA
- ☐ MarkInfo for MARKETING\_INFO
- ☐ Multidoc for MULTI\_DOC
- ☐ Property for PROPERTY
- ☐ PropAdd for PROPERTY\_ADDRESS
- ☐ PropLog for PROPERTY\_LOG
- ☐ PropALog for PROP\_AD\_LOG

Notice that the *Generate MAKEFILE* check box is not checked. Because Data Access Builder is started from the Workframe environment, the make file is generated in the Workframe project.

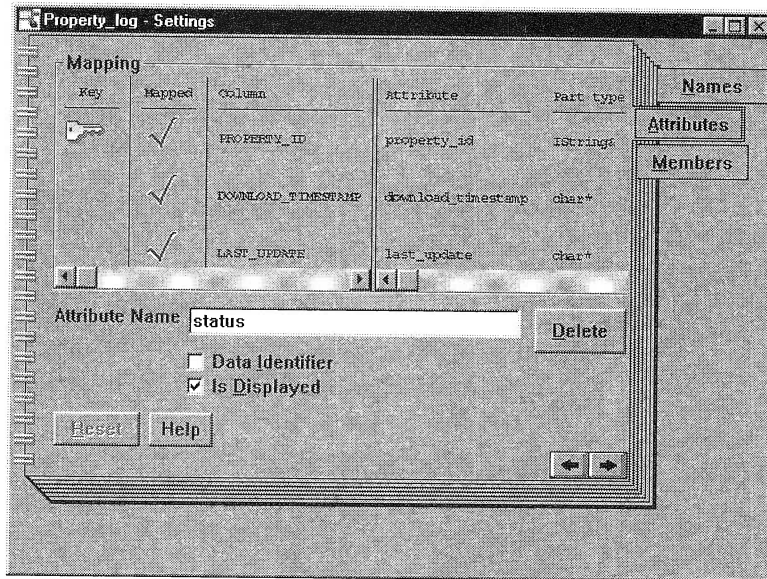
We use the Automate link to application option in the Dacslib project. The Data Access Builder puts a `#pragma library` statement with the given library name in the generated .hpp file. Then, the linker automatically looks for this library and, if it is not found, produces a link error. Check the check box and provide the library name **vrdacls** for all mapped classes as shown in Figure 46.



**Figure 46.** Property Settings Window

If you use Data Access Builder to generate one target for each table mapping, make sure that the file names are unique for each target generated. In our example, if you generate one DLL for each class, you should avoid confusing `Property`, `Property_address`, and `Property_log`. Each of them would generate a make file that would produce the same targets: `Property.dll` and `Property.lib`. In addition, the maximum length for a database package is eight characters in DB2 2.1. Thus, you should avoid file names with more than eight characters.

The *Data identifier* (Figure 47) is used to identify a row. Before the update, delete, and retrieve operations, the unique values of the data identifiers must be set to locate the row in the relational table. When the *Data identifier* check box of an attribute is selected, that attribute is used to identify a row in the table. By default, each primary key is an identifier. If the table does not have a primary key, the first attribute is selected as the default data identifier. You must ensure that the attribute contains unique values. If the values in a data identifier identify more than one row, errors occur during the retrieve operation, and multiple rows are affected during the update and delete operations. Table 6 lists the identifiers for each relational table.



**Figure 47.** Data Access Builder Attributes Page of the Settings Window

**Table 6.** Relational Table Identifiers

Table	Identifier
LIST_AREA	AREA
MARKETING_INFO	PROPERTY_ID
MULTI_DOC	MULTIDOC_ID
PROPERTY	PROPERTY_ID
PROPERTY_ADDRESS	ADDRESS_ID
PROPERTY_LOG	PROPERTY_ID
PROP_AD_LOG	PROPERTY_ID

Select **Generate** on the pop-up menu of each part to generate the C++ code in the Dacslib project folder.

Notice that, after the source code has been generated for a part, the jigsaw puzzle icon added underneath the blue ball icon is colored yellow. Later, these icons remind you that the code has already been generated and the shape represents the class type: C++ class, SOM class or Visual Builder part. Close Data Access Builder and save the session under the REAL.DAX file name.

You can now build the library and the corresponding DLL, using the *Build normal* option from your project folder. Notice that you can also generate the library from the *Build normal* option of the Property project because the Dacslib Project is embedded in the Property project.

### Warning



When building the library, make sure the database manager is already started, to prevent SQL precompiler failure when connecting to the REAL database.

## Parts Produced

When you map a table, T, to a part, several parts are generated. By default, their names start with the capitalized name of the table.

- ☐ T, a persistent object class that represents a row in the table
- ☐ TManager, a persistent object manager class that represents a collection of persistent objects T
- ☐ TDataId, a class that contains one or multiple columns that uniquely identify a persistent object
- ☐ TDatastore, a class that represents the datastore that T belongs to
- ☐ TManagerTemplate, a persistent object manager class that you use to represent a collection of persistent objects derived from T
- ☐ TManagerBase, an abstract base class for the persistent object manager classes

You use only the T and TManager in the application. The others are needed for more sophisticated tasks. The part T is derived from the IPersistentObject class and represents a row of the T table. Using the T part, you can access the information of the table, because each column is mapped to a corresponding part attribute (Data Access Builder handles type conversion between the data types defined in the database and C++). Data Access Builder generates methods to get and set the value of each attribute as well as check or set the attributes to NULL. Those attributes are enabled for notification (see Chapter 10, “More about Visual Builder...,” on page 353). In this way, attributes can be connected to other visual parts, with each attribute reflecting the change to the other parts. For each attribute, there is also an *AttributeAsString* method that returns an IString() representation. In addition, the T part supports the actions you usually apply on a table row: add, delete, update, and retrieve.



Before using these methods, you must indicate the attribute you will use to retrieve the entire row (see Figure 47 and Table 6) and then check the corresponding **Data identifier** check box. You can select several attributes as data identifiers, to identify the row by a combination of these attributes.

The TManager part is derived from IPOManager and accesses multiple rows of data. Using this part, you can retrieve several rows of the table. Use the *Refresh* method to retrieve all rows of the table, and use the *Select* method to retrieve a selected set of rows according to an SQL clause.

The rows are maintained through an attribute of type IVSequence<T\*>\*, called *iItems*. As you will see in the section below, the *iItems* attribute is used through attribute-to-attribute connections with other visual parts, such as a container or a list box, to display the contents of a set of rows.

In the current release of Data Access Builder, you must use only the *Select* method of the manager part to limit the number of rows that are read from a table and added to the IVSequence of the Manager part.

All database access is executed through the exception handler framework. In this way, exceptions are thrown by the parts whenever an error occurs, and your application can catch the exceptions to react accordingly.

## Using Data Access Builder Parts with Visual Builder

For each mapping, Data Access Builder produces a Visual Builder export file (VBE extension), which is used to import the parts definition in Visual Builder.

In the Visual Builder main window select ***Import part information...*** from the *File* menu item to import all of your VBE files. For each VBE file, a VBB file is created that consists of six parts. If you want to reorganize your files, you can move parts from one VBB file to another. We recommend that you move all Data Access Builder parts in a VRDACS.VBB file:

1. Start **Visual Builder** from the Dacslib project by selecting **Project** → **Visual** menu.
2. Select **Import part information** from the **File** menu.
3. Select all .vbe files and click on **OK**.

4. For each .vbe file, a message window opens and displays any import errors and successful completion. Close all message windows.
5. Select all newly loaded part files only.
6. Select all nonvisual parts.
7. Select the **Move** menu item either from the pop-up menu or the **Part** menu.
8. Enter the new part file name **vrdacls.vbb** in the Move parts window and select **OK**.
9. Close the Visual Builder.

You must now have a VRDACS.VBB file in your Dacslib project.

The generated parts are subclasses from parts provided by the Data Access Builder in the file VBDAX.VBB. This file is located in the D:\IBM-CPPW\IVB\ directory. You must load it in Visual Builder to successfully generate code for parts that use your data classes.

The IDatastoreBase part gathers many of the services you need to establish and manage database connection. The three derived classes (IDatastore, IDatastoreDB2, and IDatastoreODBC) provide the specific implementations for the access methods embedded SQL, DB2 CLI and ODBC CLI.


You also find two visual parts (IDSCConnectBaseCanvas and IDSCConnectCanvas) in this file that you can use to build a simple prototype of user interface for selecting a database and providing authentication information.

Now that you are a little more familiar with Data Access Builder, let us build a simple application (Figure 48) which displays the records of the PROPERTY table in a list box. This time, you do not use the Workframe environment to organize your code. Instead, you start Data Access Builder from a command prompt. To create a database connection, follow these instructions:


1. Open a command prompt session and create a directory (for example D:\TEST).
2. Copy VRDACS.LIB, VRDACS.DLL and PROPERTY.HPP in D:\TEST (these three files should be located in D:\VR\BIN). You need the LIB file for link-editing your sample application, the DLL file for executing the application and the HPP file for the compiler when compiling the application which uses the PropertyManager class.
3. Start Visual Builder from D:\TEST by issuing the ivb command. The Visual Builder window is displayed, and the working directory is set to the current directory.

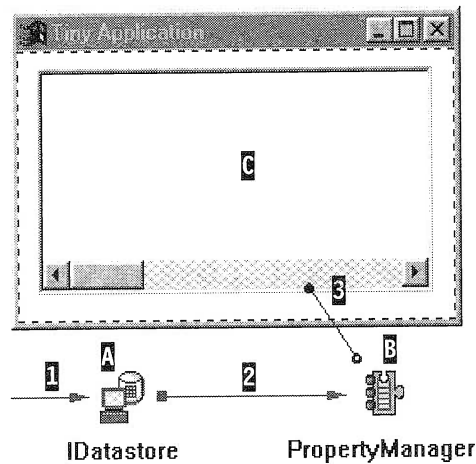
4. Load `D:\IBMCPW\IVB\VBDAX.VBB` and `D:\VR\Source\Dacs-lib\DACS.VBB` in the Visual Builder.
5. Create a new visual part with `IFrameWindow` as the base class:
  - From the Visual Builder window, select **Part** → **New...** option.
  - Fill in the entry fields as follows:

Field	Value
Class name	TinyApp
Description	Sample application with DACS
File name	TinyApp
Part type	visual part
Base class	IFrameWindow

- Click on the **Open** push button. An `IFrameWindow*` part is displayed on the free-form surface.
6. Add an `IDatastore*` part on the free-form surface,  (use **Option** → **Add parts...** from the Composition Editor menu bar).
  7. Open its settings notebook and set its attributes as follows (we assume that a user, **userid**, with a password, **password**, can connect to the **REAL** database):

Field	Value
<code>dataStoreName</code>	REAL
<code>userid</code>	userid
<code>authentication</code>	password

8. Connect the **Ready** event of the application to the **Connect** action of `IDatastore`, . (The event is accessible from the action list box displayed by selecting the **more...** option from the `IDatastore` pop-up menu.)



**Figure 48.** Simple Application with Data Access Builder

You can then use the database connection to interact with the relational tables. If you want to display the contents of the `PROPERTY` table in a list box:

1. Add a `PropertyManager*` part on the free-form surface, **B** (use **Option** → **Add parts...** from the Composition Editor menu bar).
2. From the Visual Builder palette, select an `ICollectionViewListBox*` part in the *Lists* category and drop it on the `IFrameWindow*` part. (An `ICollectionViewListBox` part is a general-purpose list box that displays objects of any type in a collection.)
3. Open the settings notebook of the `ICollectionViewListBox*` part and on the General page set the *Item type* to **Property\***, **C**. The *items* attribute maintained by the collection list box is set to the `IVSequence<Property*>` type and matches the type of the items attribute held by `PropertyManager` (see the attribute-to-attribute connection **3** in Figure 48).
4. Connect the **Connected** event of `IDatastore` to the **Refresh** action of `PropertyManager`, **2**.
5. Connect the *items* attribute of `PropertyManager` to the *items* attribute of the list box, **3**.

6. Save your part and generate its code:
  - To generate the source code of the part, select the **Save and Generate** → **Part source** option from the **File** pull-down menu of the Visual Builder window.
  - To generate the make file of your tiny application, select the **Save and Generate** → **main() for part** option from the **File** pull-down menu of the Visual Builder window.

### Hackers



When you drop a part on the free-form surface by using **Options** → **Add Parts...**, you cannot enter a part's class name without a trailing asterisk or star (\* for the dereferencing operator). If you omit the star, you can drop only a variable of the part. Also, if you drop a part from the palette, you may notice that its type is a pointer to the part itself. In fact, you cannot drop on the free-form surface a part that is not a pointer.

The relationship between a part and its subparts is an association of *containment by reference*; that is, the class of the subpart is not embedded in the class of its composite part. Rather, a pointer on the subpart is defined as an attribute of the composite part. This subtle difference facilitates development of the parts because you do not have to provide a copy constructor. However we strongly recommend that you provide one for all of your parts (see item 11 of *Effective C++* from Scott Meyers).

When you make connections between parts, you define a class that contains methods requiring parts as parameters (*Initialize* is one of them; see Chapter 9, "Connecting the Parts," on page 247). By default, a parameter is passed to a C or C++ function by value. Thus, the parameter is copied into the stack before the call and restored when the function terminates.

If you want to pass a part as a parameter, you must provide the part with a copy constructor. If you do not provide a copy constructor, you end up with compilation errors. To avoid this problem, each part you drop on the free-form surface must be a pointer whose base class holds a copy constructor!

You can now compile the code and run the application to see the contents of the table displayed in the collection list box:

- Switch to the command prompt window and make sure that D:\TEST is your current directory. (The compiler must have the Property.hpp file to compile and the linker must have the Vrdacs.lib to link-edit the tiny application.)

As you use the Automate link to application option, the linker looks for the Vrdacs.lib automatically. Otherwise you could also specify the Vrdacs.lib to the linker without changing the gener-

ated make file. You cannot pass file names to the linker in the environment variable `ILINK`. However, the linker is started through the compiler, so you can use the ICC environment variable by entering: `set icc=Vrdacs.lib`.

- ❑ On the command line, enter: **nmake tinyapp.mak**.
- ❑ Once the code has been compiled and linked, run the application by typing **tinyapp** on the command line.

All of the columns are displayed, separated by a period, in the list box. In “Creating a String Generator Class” on page 202, we explain how to choose the contents of the list box.

Now that the tables are mapped to nonvisual parts, let us tackle Visual Builder and begin to build the visual parts of the property subsystem.



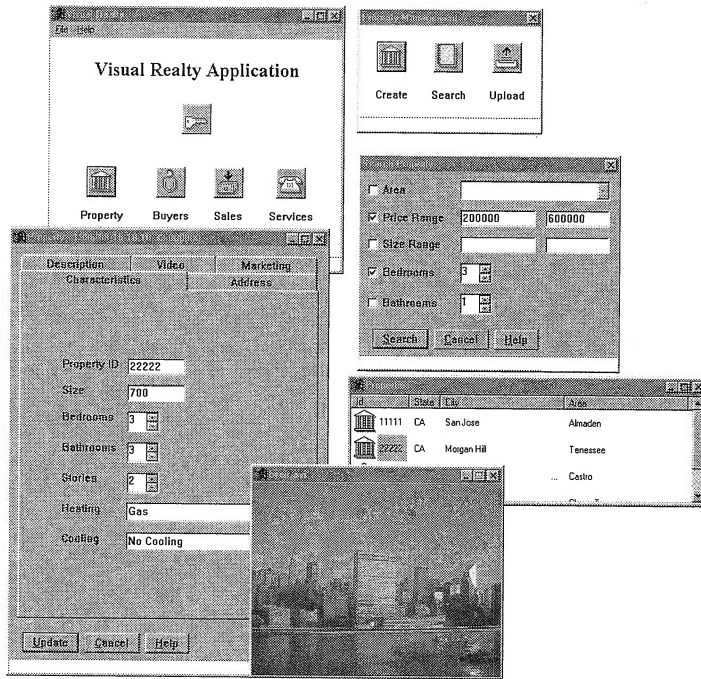
# 7

## Creating Visual Parts

A visual part is a visual representation of the application objects you defined during the design phase (see “Object Design” on page 87). The set of visual parts you define makes up the user interface of the application.

To get acquainted with the different views you are going to build, you can run the Visual Realty application that is provided as a CD-ROM with this book (see Figure 49).

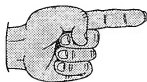




**Figure 49.** Visual Realty Application in Action

You will find all of the information you need to install and run this application in Appendix A, “Installing the Application,” on page 509. You can also consult the READ.ME file on the CD-ROM for the latest information.

#### Read This

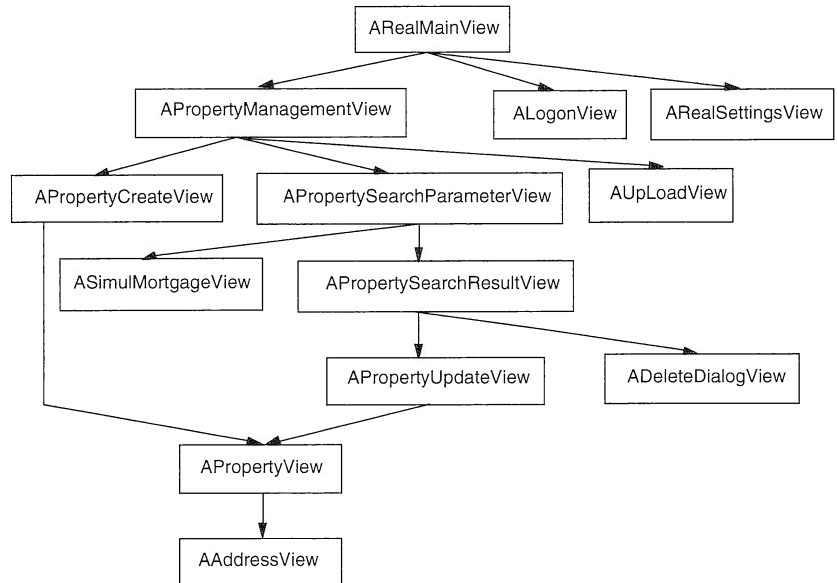


All of the visual parts that you build are prefixed with the letter A (for example, PropertyView is built as the APropertyView part). With this naming convention, the visual parts will be listed first in the Visual Builder window visual part list box.

Also, in this chapter, *view* and *composite visual part* are synonyms.

To create visual parts, we strongly suggest that you begin from the simplest parts and work up to the more complex parts. For example, to build the visual parts of the Property subsystem, start from the simplest view, AAddressView, and work up to the main view of the application: ARealMainView. You can easily apply this building process by

following the *view hierarchy structure*, which depicts the use relationship between the parts (see Figure 50). For example, according to the hierarchy, `AAddressView` is built before `APropertyView` because `AAddressView` is used by `APropertyView`, which in turn is used by `APropertyUpdateView`, and so forth.



**Figure 50.** View Hierarchy

As is evident in the view hierarchy, use of the Property subsystem requires that you build the following visual parts:

**AAddressView** displays an address. It is used in the Buyer and Property subsystems.

**APropertyView** displays the property information.

**APropertyCreateView** displays the property information when a new property is created.

**APropertyUpdateView** displays the property information when a property is updated.

**ADeleteDialogView** displays a warning message before deleting records from the database. It is reused by the Buyer, the Property, and the Sale transaction subsystems.

**ASimulMortgageView** provides the user with a simple mortgage simulation to estimate an affordable property price.

---

**APropertySearchResultView** displays a list of properties that match the buyer's criteria.

**APropertySearchParameterView** displays and collects the buyer's criteria to select properties in the database.

**AUploadView** triggers the generation of the database export files.

**APropertyManagementView** is the primary window of the Property subsystem and provides access to the property management options.

**ALogonView** collects the user's authentication to establish a database connection.

**ARealSettingsView** enables the user to update the application settings.

**ARealMainView** is the main window application and enables the user to log on to the database and to access the different subsystems.

In the sections that follow, we describe how to use Visual Builder to build the visual parts. We assume that you have some basic knowledge of the tool (for a Visual Builder "crash course," refer to "Using Visual Builder" on page 24) and are familiar with the Windows environment. We also assume that the following files are loaded in Visual Builder:

- ☐ VBDAX.VBB, database access parts
- ☐ VBMM.VBB, multimedia parts
- ☐ VBSAMPLE.VBB, general-purpose parts
- ☐ KBDHDR.VBB, general-purpose event handler for the keyboard

If you have installed VisualAge C++ on your D drive and you also have installed the samples component, you will find the first three files in the D:\IBMCPWP\IVB directory. The last file is provided on the CD-ROM that accompanies this book. Refer to "Setting Up a Project for Visual Builder" on page 121, where you will find instructions for building the VBLOAD.DAT file to load automatically these files in Visual Builder when starting Visual Builder within each subproject.

Make sure that you read Chapter 5, "Setting Up the Development Environment," on page 105 to get acquainted with the WorkFrame/2 development environment, which serves as a base for organizing our sample application. From each subproject (property, service, common, and main) that you define, you can access the Visual Builder window by selecting the **Visual** action from the Project pull-down menu or the project pop-up menu. You can also use the accelerator keys **Ctrl+Shift+V**.

---

When you construct your visual parts, you will have to configure some parts, such as set canvas or multicell canvas, to enable them to evenly display, when they resize, the controls they contain. These controls are also called *child windows*. The settings for these canvases assume that you use the default font for your controls: System Proportional - 10. Because our application has been developed on an SVGA resolution machine (1024 x 768 pixels in 256 colors), your panels may not look exactly the same if you run the application on a machine that has a different resolution.

Also, when you add an entry control, such as an entry field or a list box, to another part or the free-form surface, the number of characters you can type for the control has a default value. When you change the default, the control is not resized accordingly, except when it is dropped on a multicell canvas. You can change the width of the entry control to reflect its actual limit by selecting the *Reset to default size* option in its pop-up menu. For example, suppose the entry field control you drop on the free-form surface has a default limit of 32 characters—this limit is kept in the *limit* attribute. When you change the default to 10 characters, the width of the entry field control is not updated accordingly. To resize the controls, select the control and click on it with the right mouse button to display its pop-up menu. Then select the *Reset to default size* option to adjust its width to 10 characters.

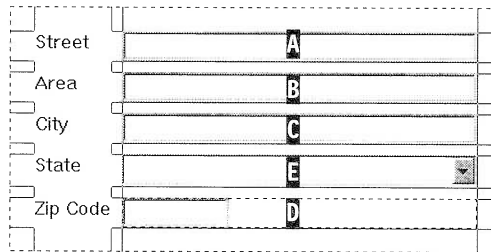
In the sections that follow, you build the visual parts of the Visual Realty application by using entry controls that hold data. The data width—the specific number of characters—is given by the corresponding attribute of its relational table (see Appendix C, “Database Definition,” on page 515). Make sure that you reset the control’s size to the default size after changing the limit of characters it can accept. As mentioned above, this constraint does not apply when you drop a control in a multicell canvas. In this case, setting the limit of the control automatically sets its width.

Finally, “adding a part” to the Composition Editor means selecting the part from its palette category and dropping it on the free-form surface or on another part. The first time you have to add a part that you have not used, we will give you its category name. This rule always applies except when we tell you to use **Option** → **Add part...** from the menu bar of the Visual Builder window.

It is now time to build your first visual part.

## AAddressView

AAddressView represents the primary view for the Address class. It consists of a multicell canvas—IMultiCellCanvas part is the name of the part that you use in Visual Builder—on which you lay out several entry fields—IEntryField part—to display the *Street*, *Area*, *City*, and *ZipCode* attributes of the Address class and a combo box—IComboBox part—to display its *State* attribute (see Figure 51).



**Figure 51.** AAddressView Part

Instead of using a standard ICanvas part, you build AAddressView as a subclass of the IMultiCellCanvas part. This canvas enables the controls that you drop onto it to expand or shrink when the canvas is resized.

An IMultiCellCanvas part is a set of cells organized in rows and columns. A multicell canvas is like a spreadsheet; each cell can contain a part, and a part can span multiple rows and columns. The cells are adjusted to vary the text length of the control. Multicell canvases enhance NLS and facilitate the use of the application in different graphical resolutions.

Each time you create a visual part that inherits from the IFrameWindow part, the canvas of its client area is a standard canvas, ICanvas part. The controls you place in such a canvas cannot be resized when the canvas expands or shrinks. However, multicell canvases enable your controls to resize when changing fonts or screen resolution. Thus, we recommend that you replace the ICanvas part with an IMultiCellCanvas part in all frame windows of the Visual Realty application.

You can work with a multicell canvas in two ways:

- You can choose to use a multicell canvas to enable the controls that you drop into it to resize when the user resizes the multicell canvas. For this purpose, you can select the columns and the rows that you want to be expandable in your multicell canvas. Each control that you drop in one of these rows or columns can then

grow or shrink according to the dimension of the multicell canvas. For example, suppose you have an entry field that contains the fully qualified name of a file. Let us say that this entry field can accept a name that is 45 characters long. You can add this entry field in a multicell canvas column that you can set to be expandable, and you can set the limit attribute of the entry field to 45. Then, you can resize the length of the entry field to your liking—this size is known as the *minimum size* of your control. The multicell canvas memorizes this size as the limit below which the control is cut off if the user continues to shrink the multicell canvas. (That is why a multicell canvas is also known as a *minimum size canvas*.) When users want to type a long file name, they can stretch the canvas so that the entry field stretches accordingly. You will use this facility in “Building the Video Page” on page 171.

- You can choose to use a multicell canvas to enable your application to be portable across different screen resolutions or font settings. In this case, the multicell canvas does all the work for you. All you do is drop your controls on the multicell canvas. Each time you change the font type or the label of those controls, they are resized according to the minimum size the control returns to the multicell canvas.

To adjust the size of a parent window to the size of its canvas in any type of resolution, the parent window must execute a `moveSizeToClient` action before displaying. This action requires, as a parameter, an `IRectangle` object that describes the position and the minimum size of the client. The action can be executed by triggering a custom logic connection (see “Using Custom Logic” on page 271) from the ready event of the free-form surface to the parent window:

```
target → moveSizeToClient(IRectangle(target → position(),
                                     target → client()
                                     → minimumSize()));
```

If you use a multicell canvas as the client canvas, you can adjust the size of the multicell canvas to the size of the parent window by setting its outermost columns and rows to be expandable. This is the method you will use in this book (see, for example, “ADeleteDialogView” on page 183).

**Warning**

When you resize a control by using its handles—the handles are the four small black boxes that appear on the corners of the control when you select it—make sure that you do not set the minimum size to a fixed value. If a fixed minimum size is set for the control and the control is used in combination with a minimum size canvas, it will not resize properly when the screen resolution is changed because its minimum size is not calculated at execution time. To ensure that the minimum size is calculated at execution time, open the control's settings notebook and, on the Size/Position page, select the **Calculate at execution time** radio button. This precaution is not necessary if you do not use the control in a minimum size canvas.

In a multicell canvas, the width of a given column is the width of the largest control in that column. Likewise, the height of a row is the height of the highest control in that row. Thus, you may encounter some trouble if, for example, you are seeking to drop two controls with different widths into the same column. The smaller control will adjust to the width of the larger. You can prevent the smaller control from resizing in two ways:

- ❑ Add another column in your multicell canvas and make the longest control span two columns. Use the **ALT** key to drag your control over multiple columns or rows. Then set the second column to be expandable. In this way, the control expands across the second column, and the shortest control is not resized. The problem with this approach is that you make the first control expandable even though expansion may not be needed. The second approach resolves this issue.
- ❑ Use another canvas in the multicell canvas and drop the shortest control on it. The constraint here is that you must use a minimum size canvas that manages the minimum size attribute of its controls. You can use the set canvas, the toolbar canvas, and the multicell canvas as the minimum size canvas. Whichever canvas you use, ensure that the visual part that you build will support different screen resolutions and font settings. Use an `ISetCanvas` part to add the zip code entry field in `AAddressView`. The `ISetCanvas` part is a set of cells organized in rows and columns called *decks*. It can be used to provide adjustable cells in rows or columns within canvases.


A final word of advice: When you use a multicell canvas, do not place controls in the first and last rows or in the first and last columns. Instead use this space for the left and right and top and bottom margins. You can set the columns and rows to be expandable so that the controls in the multicell canvas remain centered when it is resized.

To build AAddressView, follow the steps in Table 7.

<b>Table 7.</b> (Part 1 of 2) Constructing AAddressView Part													
<b>Step</b>	<b>Action</b>												
1	Start Visual Builder from the Common project (select <b>Project</b> → <b>Visual</b> in the menu bar).												
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.												
3	<p>Fill in the entry fields:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Class name</td><td>AAddressView</td></tr> <tr> <td>Description</td><td>General-purpose address view</td></tr> <tr> <td>File name</td><td>VRCOMM</td></tr> <tr> <td>Part type</td><td>visual part</td></tr> <tr> <td>Base class</td><td>IMultiCellCanvas</td></tr> </table> <p>Click on the <b>Open</b> push button. An IMultiCellCanvas* part is displayed on the free-form surface.</p>	Field	Value	Class name	AAddressView	Description	General-purpose address view	File name	VRCOMM	Part type	visual part	Base class	IMultiCellCanvas
Field	Value												
Class name	AAddressView												
Description	General-purpose address view												
File name	VRCOMM												
Part type	visual part												
Base class	IMultiCellCanvas												
4	<p>Open the settings notebook of the IMultiCellCanvas* part and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 11</li> <li><input type="checkbox"/> Number of columns: 5</li> <li><input type="checkbox"/> Expandable rows: Nos. 1, 11</li> <li><input type="checkbox"/> Expandable columns: Nos. 1, 5</li> </ul> <p>Close the settings notebook.</p>												
5	Select the <b>Sticky</b> check box, <b>A</b> , below the palette. When the <i>Sticky</i> check box is selected, the mouse pointer remains loaded with the last part dropped on the free-form surface. This is a convenient way of dropping several parts of the same type without moving the mouse pointer back and forth from the palette to the free-form surface.												
6	Add five IStaticText* parts to the IMultiCellCanvas* part, as shown in Figure 51 on page 152, and name them appropriately. Change their text attributes as follows: Street, Area, City, State, and Zip Code. (The IStaticText* part is located in the <i>Data entry</i> category.) Notice that, with the <i>Sticky</i> check box selected, the IStaticText* part remains loaded in the mouse pointer after it has been dropped. Because you are going to add four entry fields, keep the <i>Sticky</i> check box selected.												
7	Add four IEntryField* parts ( <b>A</b> , <b>B</b> , <b>C</b> , <b>D</b> ) to cells (2, 4), (4, 4), (6, 4), and (10, 4) of the IMultiCellCanvas* part. (The IEntryField* part is located in the <i>Data entry</i> category.)												



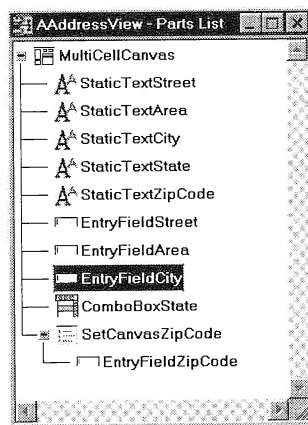
**Table 7.** (Part 2 of 2) Constructing AAddressView Part

Step	Action															
8	<p>Open the settings notebook of each IEntryField* part and set the names and limits as follows:</p> <table><tr><th>Part</th><th>Name</th><th>Limit</th></tr><tr><td>A</td><td>EntryFieldStreet</td><td>40</td></tr><tr><td>B</td><td>EntryFieldArea</td><td>40</td></tr><tr><td>C</td><td>EntryFieldCity</td><td>40</td></tr><tr><td>D</td><td>EntryFieldZipCode</td><td>5</td></tr></table> <p>Close the settings notebooks. Refer to Appendix C, “Database Definition,” on page 515 for the structure of the PROPERTY_ADDRESS table. When you add a part to another part or to the free-form surface, Visual Builder automatically assigns a name to the part. It is good practice to rename the parts to more meaningful names if you want to refer to them from other parts. You change the part’s name from its settings or from its pop-up menu.</p>	Part	Name	Limit	A	EntryFieldStreet	40	B	EntryFieldArea	40	C	EntryFieldCity	40	D	EntryFieldZipCode	5
Part	Name	Limit														
A	EntryFieldStreet	40														
B	EntryFieldArea	40														
C	EntryFieldCity	40														
D	EntryFieldZipCode	5														
9	In Step 10, you add only one IComboBox* part in the IMultiCellCanvas* part. Thus, you can now deselect the <i>Sticky</i> check box.															
10	Add an IComboBox* part to cell (8, 4) of the IMultiCellCanvas* part. (The IComboBox* part is located in the <i>Lists</i> category.)															
11	<p>Open the settings notebook of the IComboBox* part, , and set its type as <b>read-only drop-down</b> and its name and limit as follows:</p> <table><tr><th>Part</th><th>Name</th><th>Limit</th></tr><tr><td>E</td><td>ComboStateBox</td><td>20</td></tr></table> <p>Close the settings notebook. Refer to Appendix C, “Database Definition,” on page 515 for the structure of the PROPERTY_ADDRESS table.</p>	Part	Name	Limit	E	ComboStateBox	20									
Part	Name	Limit														
E	ComboStateBox	20														
12	Add an ISetCanvas* part to cell (10, 4) of the IMultiCellCanvas* part. This canvas will hold the entry field for the zip code. (The ISetCanvas* part is located in the <i>Composers</i> category.)															
13	Open the settings notebook of the ISetCanvas* part and, on the General page, set the width and height of the margin to <b>0</b> . Then close the settings notebook.															
14	Select the IMultiCellCanvas* part and from its pop-up menu select the <b>Reset to default size</b> option to adjust its size to the controls it holds.															
<b>Note:</b> Reverse highlighted letters are keyed to Figure 51 on page 152.																

You can improve your visual part so that the user can use the keyboard to move the input focus from one subpart to another.

## Setting the Tabbing and Depth Order

The depth order within a composite part, such as AAddressView, is the order in which parts are stacked on the application desktop. Visual Builder assigns the depth order as parts are dropped. This order can be checked and changed from the parts list which is displayed by selecting the **View Parts List...** option from the free-form surface or from any composite part's pop-up menu. For example, you build AAddressView by first dropping five static text controls on the multicell canvas. The parts list you can then display by selecting the **View Parts List...** option from the free-form surface shows the static text controls in first position after the multicell canvas (Figure 52).



**Figure 52.** AAddressView Parts List

From the parts list you can:

- ☐ Change the position of parts to reflect their order in the Composition Editor. In effect, the order in which parts are placed on a canvas part determines their tabbing order. You probably need to change the order of the list as you add or rearrange parts. You move one line of the list by dragging and dropping it to the location of your choice.
- ☐ Perform operations on parts as you do in the Composition Editor. You can access the pop-up menu of a part from the order list. You will find this operation useful when you want to access a part that you cannot see in the Composition Editor.

The tabbing order is the order in which the input focus moves from one part to another as the user presses the Tab key. The tabbing order is also the order in which the input focus moves among parts within a tab group (for example a group of radio buttons) as the user presses the arrow keys. The initial tabbing order is determined by the order in

which you place the parts on the Composers part (the initial depth order). Also, the first part in the tabbing order receives the initial input focus. For example in AAddressView, the EntryFieldStreet receives the input focus first.

To activate the tabbing order feature, you must set tab stops for every part you want to receive the input focus as the user presses the Tab key. For example to set a tab stop for EntryFieldStreet:

1. Select EntryFieldStreet.
2. Open the part's pop-up menu with the right mouse button.
3. Select **Set tabbing** → **Tab stop**.

Repeat these three steps for the other entry fields and the drop-down list box.

If you want the user to be able to move the input focus to a part with the keyboard arrow keys, you must define a group of parts by selecting the **Group** check mark from the **Set tabbing** option of the first part's pop-up menu in the group—this part is called the *group part*. For example, to allow the user to move the input focus in AAddressView by using the arrow keys, you define EntryFieldStreet as the first group part:

1. Select EntryFieldStreet.
2. Open the part's pop-up menu with the right mouse button.
3. Select **Set tabbing** → **Group**.

To start another group, select **Set tabbing** → **Group** for the part that you want to be the first part in the group. If a part has both Group and Tab stop selected, a user can tab to the first part in the group and then use the arrow keys to move to the other parts in the group.

AAddressView is reused by APropertyView to display the address of a property. Therefore, APropertyView must have access to the contents of each entry field of AAddressView and to the contents of its combo box.

## Promoting a Part Feature

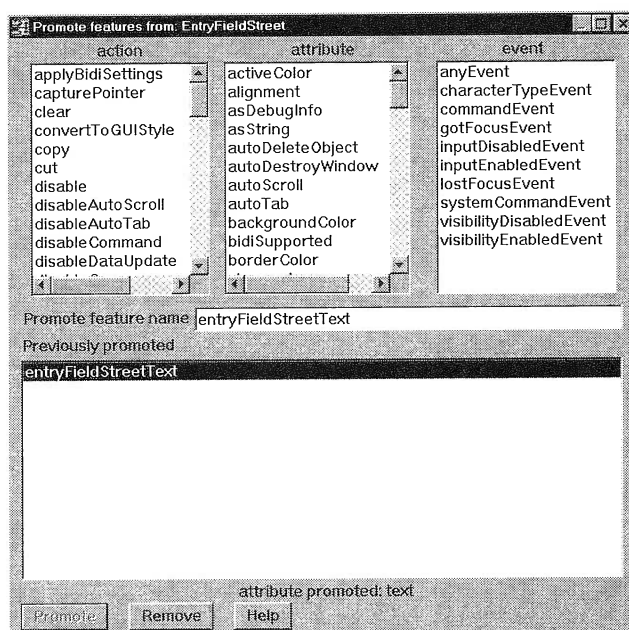
Promoting a part feature is a way of exposing the feature to another part making it available through that other part. When a feature is promoted in part A, it can be accessed from another part, B, when part A is embedded as a subpart within part B. Thus, to use AAddressView as a subpart and access the contents of its entry fields and its combo box in another part, you must promote the *text* attribute of these controls.

When you define a part in Visual Builder, the new features you add to the part are not available from other parts unless you promote them. You can promote the feature of a part in two ways:

- ❑ Select the part and use the **Promote part feature...** option from the part's pop-up menu.
- ❑ Use the Promote page in the Class Editor.

Let us promote the *text* attribute of EntryFieldStreet:

1. Select EntryFieldStreet (A in Figure 51 on page 152) and click on it with the right mouse button.
2. Select the **Promote part feature ...** option.
3. Promote the *text* attribute (Figure 53).



**Figure 53.** Promote EntryFieldStreetText Attribute of AAddressView

Now that you have promoted your first feature, you must promote the *text* attribute of the other controls: ComboBoxState, EntryFieldArea, EntryFieldCity, and EntryFieldZipCode.

When you have promoted all features, switch to the Class Editor and fill in the *Code generation file* group box as follows:

- ❑ C++ header file (.hpp): **vrcadv.hpp**

- ❑ C++ code file (.cpp): `vrcdrv.cpp`

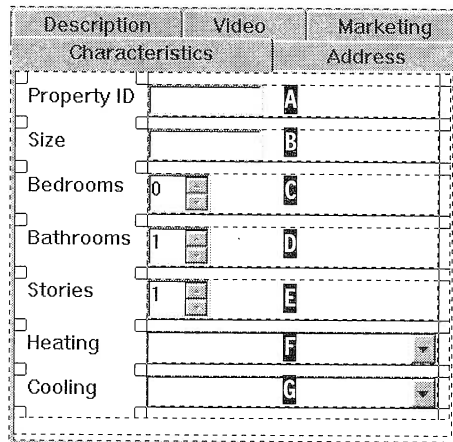
Save your part. At this time it is not necessary to generate the code because the connections have not been drawn.

## APropertyView

APropertyView is the view of the Property class. In the design object model of the Property subsystem (Figure 40 on page 100), the Property class is represented as an association of four different classes:

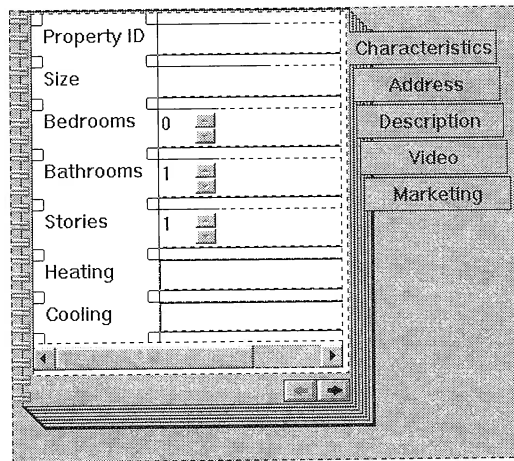
- ❑ Address
- ❑ MarketingInfo
- ❑ PropertyLog
- ❑ MultiDoc

To reflect this association, you design the view of Property as a notebook whose pages represent the respective view of each class component. In Figure 54, the notebook is shown using its native representation in Windows. This representation is also called *property sheet* or *tab dialog box* in the Microsoft Foundation Class Library dialect.



**Figure 54.** Windows Version of APropertyView Part

If you prefer, you can switch to the Common User Access version (CUA is an IBM architecture for designing graphical user interfaces using a set of standard components and terminology) of the notebook by setting the *pmCompatible* style to *On* in the Styles page of your notebook's settings notebook (see Figure 55 for the CUA representation of a notebook).



**Figure 55.** CUA Version of APropertyView Part

### Portability



Make sure that the `pmCompatible` style is set the way you want it:

- If you select the **On** radio button, you have access only to functions that are common to OS/2 and Windows. The compiled notebook looks like a CUA notebook, even in Windows.
- If you select the **Off** radio button, you have access to all functions in the native notebook. These functions vary between OS/2 and Windows. In OS/2 the compiled notebook looks like a CUA notebook. In Windows, the notebook looks like a native Windows property sheet.

The `PropertyLog` class is not represented as a notebook page because it does not have any visual representation. It is used to hold the time stamp of each creation or update in the database. In addition, a `Description` page is added to display information that is related to the `Property` class but does not fit on the `Property` page.

In the sections that follow, you build `APropertyView` by using a Windows version of the notebook (`pmCompatible` set to **Off** or **Default** in the `Styles` page of the settings notebook) and then enhance it with the multicell canvas and the viewport parts.

**Read This**

You must promote all field attributes of the APropertyView part because the part is reused in APropertyCreateView and APropertyUpdateView, and these attributes must be accessible.

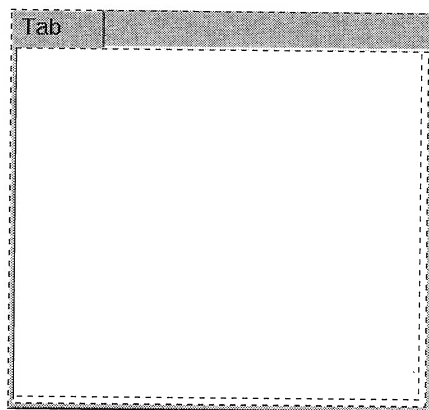
## Using a Notebook Control

The INotebook\* part is a software representation of a physical notebook. It presents information on tabbed pages that the user can display sequentially or randomly. For an example of a notebook control, open any Visual Builder settings editor.

When building a notebook, you can use its various settings to tailor its appearance. You can:

- ☐ Select the type of binding (CUA version only).
- ☐ Define the tab appearance, tab size, and tab text alignment.
- ☐ Define the page button size.

Once you have tailored the appearance of the notebook, you can add pages by selecting an add page choice from its pop-up menu. If the notebook does not have any pages, your only choice is to add a first page with the *Add Initial Page* option. If the notebook has one or more pages, select a page and choose **Add Page After** or **Add Page Before**. When a notebook is added to the Composition Editor, it already has one page (Figure 56).



**Figure 56.** Windows Version Notebook for APropertyView

To build APropertyView as a notebook, follow the steps in Table 8.

<b>Table 8.</b> Building APropertyView As a Notebook													
<b>Step</b>	<b>Action</b>												
1	Start Visual Builder from the Property project (select <b>Project</b> → <b>Visual</b> in the menu bar).												
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.												
3	<p>Fill in the entry fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Class name</td><td>APropertyView</td></tr> <tr> <td>Description</td><td>Property primary view</td></tr> <tr> <td>File name</td><td>VRPROP</td></tr> <tr> <td>Part type</td><td>visual part</td></tr> <tr> <td>Base class</td><td>INotebook</td></tr> </table> <p>Click on the <b>Open</b> push button. An INoteBook* part is displayed on the free-form surface with one initial page.</p>	Field	Value	Class name	APropertyView	Description	Property primary view	File name	VRPROP	Part type	visual part	Base class	INotebook
Field	Value												
Class name	APropertyView												
Description	Property primary view												
File name	VRPROP												
Part type	visual part												
Base class	INotebook												
4	<p>Open the settings notebook of the INotebook* part and set its appearance as follows:</p> <table> <tr> <th>Setting</th><th>Value</th></tr> <tr> <td>Status area</td><td>Left</td></tr> <tr> <td>Tab</td><td>Center</td></tr> </table> <p>Close the settings notebook.</p>	Setting	Value	Status area	Left	Tab	Center						
Setting	Value												
Status area	Left												
Tab	Center												
5	Add four more pages, using the <b>Add Page</b> → <b>After Top Page</b> option.												
6	Switch to the Styles page and set <b>allTabsVisible</b> to <b>On</b> . This setting forces the notebook to display all the tabs of your notebook pages.												

## Building the Pages of a Notebook

You can construct a notebook page by using one of two methods:

- ❑ Assemble controls on a separate canvas and drop the canvas on the page client area. Use this method if the visual part for the notebook page is reusable. For example, you built the AAddressView part (Figure 51 on page 152) as a reusable part and can drop it on the Address page (Figure 59 on page 169). It is good practice to build your notebook page on a separate canvas, especially when the pages are complex. Furthermore, you can encapsulate nonvisual parts at the page level.



- ❑ Assemble controls directly on the notebook page. Use this method if the visual part for the notebook page is not reusable or the page is simple enough to be built directly in the notebook. Because the Marketing, Characteristics, Video, and Description pages of our Visual Realty application are fairly simple, you can build them directly in the notebook (Figure 62 on page 174).

Although you build the Marketing, Characteristics, Video, and Description pages as nonreusable canvases, you should always design your parts to be reusable. You may not reuse a specific part in your application, but you never know when you will need the part in another application. Building for reuse implies building for other applications.

### ***Enhancing the Notebook Page***

When you add a page to a notebook, the page is created with an ICanvas\* part as a client area. On this canvas, you can add several parts to enhance your page. As you know, the standard canvas is suitable for most situations, but it does not allow the controls to be evenly distributed when it resizes. Thus, to enhance the notebook, you use a multicell canvas for all pages.

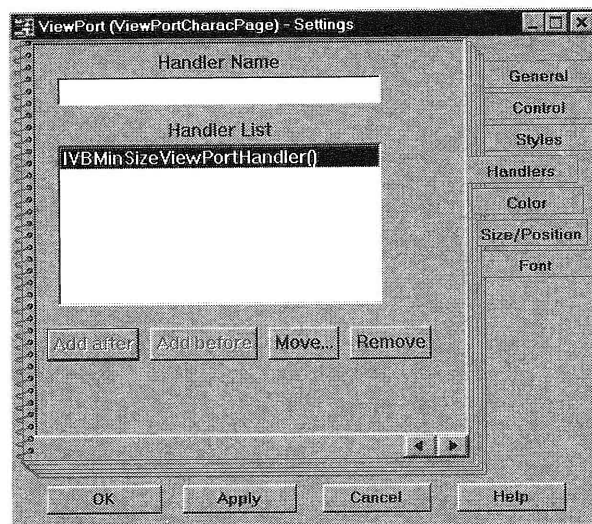
The multicell canvas is not a panacea for all resizing problems, however. In effect, every canvas, whatever its type, has a fixed minimum size that corresponds to the size of the canvas in the Composition Editor. Thus, the use of a multicell canvas for a notebook page does not prevent controls on the page from being clipped when the notebook is downsized.

In order for the user to scroll the page to access information when the page is clipped, you must use a viewport in the notebook page. The IViewport part belongs to the *Composers* category. It is a scrollable framework for any type of canvas (Figure 57). The user interacts with the controls placed inside the viewport and can scroll both horizontally and vertically if the control does not fit in the frame window.

Description	Video	Marketing
Characteristics		Address
Property ID		
Size		
Bedrooms	0	
Bathrooms	1	
Stories	1	
Heating		

**Figure 57.** Characteristics Page Using a Viewport

In the Visual Realty application, you use an `IViewPort` part for each notebook page. In addition, you must attach a specific handler, `IVBMinSizeViewPortHandler`, to each viewport. This handler ensures that when a viewport grows in size, its child part will grow with it. In our case, the child part is a multicell canvas that can to enlarge proportionally. Without the handler, the multicell canvas would not enlarge to fit the dimensions of the viewport.



**Figure 58.** Event Handler List Box

**Tip**

To select a page in a notebook, click between the two horizontal lines at the bottom of a page. The two lines provide you with visual help to locate the page within the notebook. They are not shown at run time.

### ***Building the Characteristics Page***

The characteristics page is built from basic controls such as entry fields, combo boxes, and numeric spin buttons. It serves as a good example of the use of different controls for different needs. You build this part in the same way you built AAddressView. This time, you use the ISetCanvas part for each control that you drop on the multicell canvas because the width of the control differs.

To build the characteristics page, follow the steps in Table 9.

**Table 9.** (Part 1 of 3) Building the Characteristics Page

Step	Action						
1	Select the first notebook page and change its tab label to <b>Characteristics</b> .						
2	<p>Open the settings notebook of the INotebook* part and set the tab parameters as follows:</p> <table> <tr> <th>Setting</th><th>Value</th></tr> <tr> <td>Major tab width</td><td>130</td></tr> <tr> <td>Major tab height</td><td>30</td></tr> </table> <p>Close the settings notebook.</p> <p>The tab length is adjusted to fit the largest label of the notebook. Usually you choose the tab length after you have entered its labels. Use the <b>Apply</b> push button of the settings notebook to adjust the length of the tabs.</p>	Setting	Value	Major tab width	130	Major tab height	30
Setting	Value						
Major tab width	130						
Major tab height	30						
3	Click inside the page to select the canvas and remove it.						
4	Add an IViewport* part to the page. (The IViewport* part is located in the <i>Composers</i> category.)						
5	Open the settings notebook of the IViewport* part and, on the Handlers page, add the IVBMinSizeViewportHandler handler to the handler list (Figure 58 on page 165). Close the settings notebook.						

**Table 9.** (Part 2 of 3) Building the Characteristics Page

Table 67 (Part 1 of 3) Building and Configuring the IMultiCellCanvas\* Part

Step	Action																								
6	<p>Add an IMultiCellCanvas* part in the IViewPort* part. (IMultiCellCanvas* part is located in the <i>Composers</i> category.) Configure the IMultiCellCanvas part as follows:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> Number of rows: 15</li><li><input type="checkbox"/> Number of columns: 5</li><li><input type="checkbox"/> Expandable rows: Nos. 1, 15</li><li><input type="checkbox"/> Expandable columns: Nos. 1, 5</li></ul>																								
7	<p>Add seven IStaticText* parts to the IMultiCellCanvas* part as shown in Figure 54 on page 160, and change their text attributes in this order: Property ID, Size, Bedrooms, Bathrooms, Stories, Heating, and Cooling.</p>																								
8	<p>Add seven ISetCanvas* parts in cells (2, 4), (4, 4), (6, 4), (8, 4), (10, 4), (12, 4), and (14, 4) of the IMultiCellCanvas* part.</p>																								
9	<p>Add two IEntryField* parts (A, B) to the ISetCanvas* parts in cells (2, 4) and (4, 4) of the IMultiCellCanvas* and set their names and limits as follows:</p> <table><tr><th>Part</th><th>Name</th><th>Limit</th></tr><tr><td>A</td><td>EntryFieldPropertyID</td><td>5</td></tr><tr><td>B</td><td>EntryFieldSize</td><td>5</td></tr></table>	Part	Name	Limit	A	EntryFieldPropertyID	5	B	EntryFieldSize	5															
Part	Name	Limit																							
A	EntryFieldPropertyID	5																							
B	EntryFieldSize	5																							
10	<p>Add three INumericSpinButtons* parts (C, D, E) to the ISetCanvas* parts located in cells (6, 4), (8, 4), and (10, 4) (the INumericSpinButton* part is located in the <i>Data entry</i> category) and set them up as follows:</p> <table><tr><th>Part</th><th>Name</th><th>Limit</th><th>Lower</th><th>Upper</th><th>Value</th></tr><tr><td>C</td><td>NumericSpinButtonBedrooms</td><td>1</td><td>0</td><td>6</td><td>0</td></tr><tr><td>D</td><td>NumericSpinButtonBathrooms</td><td>1</td><td>0</td><td>6</td><td>0</td></tr><tr><td>E</td><td>NumericSpinButtonStories</td><td>1</td><td>1</td><td>3</td><td>1</td></tr></table> <p>A property with no bedrooms is a studio.</p>	Part	Name	Limit	Lower	Upper	Value	C	NumericSpinButtonBedrooms	1	0	6	0	D	NumericSpinButtonBathrooms	1	0	6	0	E	NumericSpinButtonStories	1	1	3	1
Part	Name	Limit	Lower	Upper	Value																				
C	NumericSpinButtonBedrooms	1	0	6	0																				
D	NumericSpinButtonBathrooms	1	0	6	0																				
E	NumericSpinButtonStories	1	1	3	1																				
11	<p>Add two IComboBox* parts (F and G) to the ISetCanvas* parts in cells (12, 4) and (14, 4) and set their names and limits as follows:</p> <table><tr><th>Part</th><th>Name</th><th>Limit</th></tr><tr><td>F</td><td>ComboBoxHeating</td><td>20</td></tr><tr><td>G</td><td>ComboBoxCooling</td><td>20</td></tr></table>	Part	Name	Limit	F	ComboBoxHeating	20	G	ComboBoxCooling	20															
Part	Name	Limit																							
F	ComboBoxHeating	20																							
G	ComboBoxCooling	20																							

**Table 9.** (Part 3 of 3) Building the Characteristics Page

Step	Action																								
12	<p>Open the settings notebook of ComboBoxHeating and set its contents as follows:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> No Heating</li><li><input type="checkbox"/> Gas Electric</li><li><input type="checkbox"/> Propane Gas</li><li><input type="checkbox"/> Bottled Gas</li><li><input type="checkbox"/> Solar</li><li><input type="checkbox"/> Oil Central</li><li><input type="checkbox"/> Forced Air Wall</li><li><input type="checkbox"/> Furnace Floor</li><li><input type="checkbox"/> Furnace Radiant</li><li><input type="checkbox"/> Baseboard</li><li><input type="checkbox"/> Steam or Hot Water</li><li><input type="checkbox"/> Heat Pump</li><li><input type="checkbox"/> Other</li></ul> <p>Select <b>Read-only drop-down</b> in the <i>Combo box type</i> group box. Close the settings notebook.</p>																								
13	<p>Open the settings notebook of ComboBoxCooling and set its contents as follows:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> No Cooling</li><li><input type="checkbox"/> Central Conditioner</li><li><input type="checkbox"/> Room Conditioner</li><li><input type="checkbox"/> Evaporative Cooler</li><li><input type="checkbox"/> Other</li></ul> <p>Select <b>Read-only drop-down</b> in the <i>Combo box type</i> group box. Close the settings notebook.</p>																								
14	<p>For each entry part of the IMultiCellCanvas*, select the tabbing and depth order (see “Setting the Tabbing and Depth Order” on page 157):</p> <table><tr><th>Group</th><th>Tab</th><th>Feature</th></tr><tr><td>X</td><td>X</td><td>EntryFieldPropertyID</td></tr><tr><td>-</td><td>X</td><td>EntryFieldSize</td></tr><tr><td>-</td><td>X</td><td>NumericSpinButtonBedrooms</td></tr><tr><td>-</td><td>X</td><td>NumericSpinButtonBathrooms</td></tr><tr><td>-</td><td>X</td><td>NumericSpinButtonStories</td></tr><tr><td>-</td><td>X</td><td>ComboBoxHeating</td></tr><tr><td>-</td><td>X</td><td>ComboBoxCooling</td></tr></table> <p>Close the dialog box.</p>	Group	Tab	Feature	X	X	EntryFieldPropertyID	-	X	EntryFieldSize	-	X	NumericSpinButtonBedrooms	-	X	NumericSpinButtonBathrooms	-	X	NumericSpinButtonStories	-	X	ComboBoxHeating	-	X	ComboBoxCooling
Group	Tab	Feature																							
X	X	EntryFieldPropertyID																							
-	X	EntryFieldSize																							
-	X	NumericSpinButtonBedrooms																							
-	X	NumericSpinButtonBathrooms																							
-	X	NumericSpinButtonStories																							
-	X	ComboBoxHeating																							
-	X	ComboBoxCooling																							
<p><b>Note:</b> Reverse highlighted letters are keyed to Figure 54 on page 160.</p>																									

## Building the Address Page

To build the address page, reuse AAddressView as shown in Figure 59.

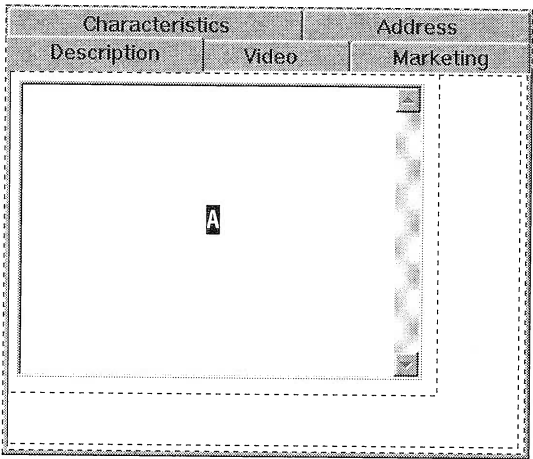
**Figure 59.** Address Page

Follow the steps in Table 10.

Table 10. Building the Address Page	
Step	Action
1	Select the second notebook page and change its tab label to <b>Address</b> .
2	Click inside the page to select the canvas and remove it.
3	Add an IViewPort* part to the page.
4	Open the settings notebook of the IViewPort* part and, on the Handlers page, add IVBMinSizeViewPortHandler to the handler list (Figure 58 on page 165). Close the settings notebook.
5	Add an AAddressView* part in the IViewPort* part ( <b>Option</b> → <b>Add parts...</b> from the Composition Editor menu), <b>A</b> . The AAddressView* part is added to the page. Notice that you cannot access its subparts.
<b>Note:</b> The reverse highlighted letter is keyed to Figure 59.	

**Building the Description Page**

The description page (Figure 60) consists of three controls: a viewport, a multicell canvas, and a multiple-line edit (MLE) control. The MLE control provides users with a basic word processor that enables them to briefly describe the property.



**Figure 60.** Description Page

To build the description page, follow the steps in Table 11.

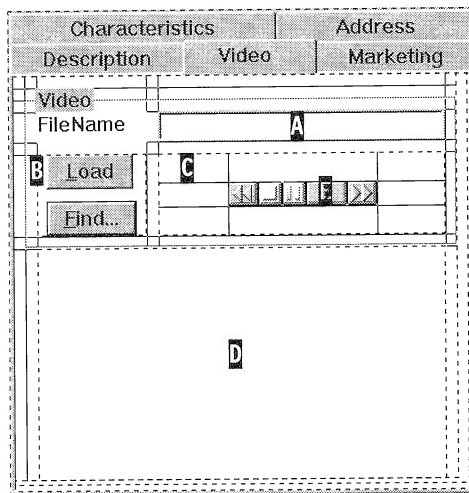
Table 11. (Part 1 of 2) Building the Description Page	
Step	Action
1	Select the third notebook page and change its tab label to <b>Descrip-tion</b> .
2	Click on the page to select the canvas and remove it.
3	Add an IViewPort* part to the page.
4	Open the settings notebook of the IViewPort* part and, on the Han-dlers page, add IVBMinSizeViewPortHandler to the handler list (Figure 58 on page 165). Close the settings notebook.

**Table 11.** (Part 2 of 2) Building the Description Page

Step	Action
5	<p>Add an IMultiCellCanvas* part to the IViewPort* part. Stretch the part to fill in the page and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 3</li> <li><input type="checkbox"/> Number of columns: 3</li> <li><input type="checkbox"/> Expandable row: No. 2</li> <li><input type="checkbox"/> Expandable column: No. 2</li> </ul> <p>The MLE control will grow when the page is resized.</p>
6	<p>Add an IMultiLineEdit* part to the IMultiCellCanvas* part (IMultiLineEdit* part is located in the <i>Data entry</i> category) and change its name to MultiLineEditDescription, <b>A</b>. Stretch the part to fill in the page.</p>
<b>Note:</b> The reverse highlighted letter is keyed to Figure 60.	

### Building the Video Page

The video page (Figure 61) enables the buyer to watch a video of the property. The page is built from a visual representation of a VCR command control: the IMMPlayerPanel part. For this part to be added to the Composition Editor, the VBMM.VBB file must be loaded in Visual Builder (**File** → **Load** from the Visual Builder window).

**Figure 61.** Video Page








This time, you build the part by using several multicell canvases, each of which is in charge of handling the resizing of a specific portion of the page. On the Video page, the *file name entry* field and the canvas where the video is displayed are expandable. A group box is used to logically group the video controls and the entry field.

To build the video page, follow the steps in Table 12.

<b>Table 12.</b> (Part 1 of 3) Building the Video Page	
<b>Step</b>	<b>Action</b>
1	Select the fourth notebook page and change its tab label to <b>Video</b> .
2	Click inside the page to select the canvas and remove it.
3	Add an IViewPort* part to the page.
4	Open the settings notebook of the IViewPort* part and, on the Handlers page, add IVBMinSizeViewPortHandler to the handler list (Figure 58 on page 165). Close the settings notebook.
5	<p>Add an IMultiCellCanvas* part to the IViewPort* part. Stretch the part to fill in the page and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 11</li> <li><input type="checkbox"/> Number of columns: 9</li> <li><input type="checkbox"/> Expandable rows: Nos. 1, 10, 11</li> <li><input type="checkbox"/> Expandable columns: Nos. 1, 5, 9</li> </ul> <p>Use Figure 62 on page 174 to position the controls in the IMultiCellCanvas* part. Notice that Row 10 and Column 5 are set to expandable so that the entry field and the multicell canvas used for the video canvas can expand.</p>
6	Add one IStaticText* part to cell (4, 3) of the IMultiCellCanvas* part and change its text attribute to <b>Filename</b> .
7	Add one IEntryField* part to cell (4, 5) of the IMultiCellCanvas* part and set its name to <b>EntryFieldVideo</b> and its limit to 40, <b>A</b> .
8	Open the settings of EntryFieldVideo and, on the Styles page, set the <i>readOnly</i> radio button to <b>On</b> to prevent the user from entering a file name in the entry field. To set the contents of the entry field, the user must use the <b>Find...</b> push button.
9	Add an ISetCanvas* part, <b>E</b> , to cell (6, 3) of the IMultiCellCanvas* part. This canvas will contain two push buttons.
10	Open the settings notebook of the ISetCanvas* part and, on the General page, set the deck orientation to <b>Vertical</b> and the margin width to <b>0</b> . The push buttons dropped onto the canvas will line up vertically.

**Table 12.** (Part 2 of 3) Building the Video Page

Step	Action
11	<p>Add an IMultiCellCanvas* part, , to cell (6, 5) of the first IMultiCellCanvas* part and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 3</li> <li><input type="checkbox"/> Number of columns: 3</li> <li><input type="checkbox"/> Expandable rows: Nos. 1, 3</li> <li><input type="checkbox"/> Expandable columns: Nos. 1, 3</li> </ul> <p>This canvas will contain the IMMPlayerPanel part and will ensure that the panel remains centered underneath the entry field when the page is resized.</p>
12	<p>Add an IMultiCellCanvas* part, , to cell (10, 5) of the first IMultiCellCanvas* part and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 3</li> <li><input type="checkbox"/> Number of columns: 3</li> <li><input type="checkbox"/> Expandable row: No. 2</li> <li><input type="checkbox"/> Expandable column: No. 2</li> </ul> <p>This canvas will contain the canvas for the video and will grow when the page is resized.</p>
13	<p>Add an ICanvas* part to cell (2, 2) of the IMultiCellCanvas* part, . (ICanvas* part is located in the <i>Composers</i> category.) The video is displayed inside.</p>
14	<p>Add two IPushButton* parts to the ISetCanvas* part, , and change their labels as shown in Figure 61. (The IPushButton* part is located in the <i>Buttons</i> category.)</p>
15	<p>Open the settings notebook of the <b>Load</b> push button and set its name to <b>PushButtonLoad</b>. Close the settings notebook.</p>
16	<p>Open the settings notebook of the <b>Find...</b> push button and set its name to <b>PushButtonFind</b>. Then switch to the Styles page and set the <i>defaultButton</i> radio button to <b>On</b>. Close the settings notebook.</p>
17	<p>Add one IMMPlayerPanel* part, , to the cell (2, 2) of the IMultiCellCanvas* part (<b>Option</b> → <b>Add part...</b> from the Composition Editor menu).</p>
18	<p>Add an IGroupBox* part on the first IMultiCellCanvas* part, (The IGroupBox* part is located in the <i>Data entry</i> category.), and change its name to <b>Video</b>. Extend the group box from cell (2, 2) to cell (7, 6) to span the IEntryField* and IMMPlayerPanel* parts. (Use the <b>ALT</b> key to extend the group box beyond the multicell canvas cell boundaries.)</p>

**Table 12.** (Part 3 of 3) Building the Video Page

Step	Action												
19	<p>Modify the Tabbing and Depth Order of the push buttons and the entry field by defining two groups of controls as follows (see “Setting the Tabbing and Depth Order” on page 157):</p> <table><tr><th>Group</th><th>Tab</th><th>Feature</th></tr><tr><td>X</td><td>X</td><td>EntryFieldVideo</td></tr><tr><td>X</td><td>X</td><td>PushButtonLoad</td></tr><tr><td>-</td><td>X</td><td>PushButtonFind</td></tr></table> <p>Close the dialog box.</p>	Group	Tab	Feature	X	X	EntryFieldVideo	X	X	PushButtonLoad	-	X	PushButtonFind
Group	Tab	Feature											
X	X	EntryFieldVideo											
X	X	PushButtonLoad											
-	X	PushButtonFind											
<b>Note:</b> Reverse highlighted letters are keyed to Figure 61 on page 171.													

### Building the Marketing Page

The marketing page (Figure 62) displays the marketing information about the property. It is built from basic controls such as entry fields and static text controls. The static text controls **A**, **B**, **C**, and **D** are updated by the nonvisual part, AMarketingInfo, which computes their value according to the contents of entry fields **E**, **F**, and **G**.

Characteristics		Address	
Description	Video	Marketing	
Price	<input type="text" value="E"/>		
Price/Sqft	<input type="text" value="Not available"/>	<input type="text" value="A"/>	
Days on Market	<input type="text" value="Not available"/>	<input type="text" value="B"/>	
Commission			
Rate	<input type="text" value="F"/>		
Value	<input type="text" value="Not available"/>	<input type="text" value="C"/>	
Down Payment			
Rate	<input type="text" value="G"/>		
Value	<input type="text" value="Not available"/>	<input type="text" value="D"/>	

**Figure 62.** Marketing Page

To build the marketing page, follow the steps in Table 13.

<b>Table 13.</b> (Part 1 of 2) Building the Marketing Page											
<b>Step</b>	<b>Action</b>										
1	Select the last notebook page and change its tab label to <b>Marketing</b> .										
2	Click on the page to select the canvas and remove it.										
3	Add an <code>IViewPort*</code> part to the page.										
4	Open the settings notebook of the <code>IViewPort*</code> part and, on the <b>Handlers</b> page, add <code>IVBMinSizeViewPortHandler</code> to the handler list (Figure 58 on page 165). Close the notebook settings.										
5	<p>Add an <code>IMultiCellCanvas*</code> part to the <code>IViewPort*</code> part. Stretch the part to fill in the page and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 22</li> <li><input type="checkbox"/> Number of columns: 7</li> <li><input type="checkbox"/> Expandable rows: Nos. 1, 22</li> <li><input type="checkbox"/> Expandable columns: Nos. 1, 7</li> </ul> <p>Use Figure 62 on page 174 to position the controls in the <code>IMultiCellCanvas*</code> part.</p>										
6	<p>Add seven <code>IStaticText*</code> parts to the second column of the <code>IMultiCellCanvas*</code> as shown in Figure 62 on page 174 and set their text attributes as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Price</li> <li><input type="checkbox"/> Price/Sqft</li> <li><input type="checkbox"/> Days on Market</li> <li><input type="checkbox"/> Rate</li> <li><input type="checkbox"/> Value</li> <li><input type="checkbox"/> Down Payment Rate</li> <li><input type="checkbox"/> Value</li> </ul>										
7	<p>Add four more <code>IStaticText*</code> parts (<b>A</b>, <b>B</b>, <b>C</b>, <b>D</b>) to cells (5, 5), (7, 5), (13, 5), and (20, 5) of the <code>IMultiCellCanvas*</code> part and set their part names as follows:</p> <table border="1"> <thead> <tr> <th>Part</th><th>Name</th></tr> </thead> <tbody> <tr> <td>A</td><td>StaticTextPriceSqft</td></tr> <tr> <td>B</td><td>StaticTextDaysOnMarket</td></tr> <tr> <td>C</td><td>StaticTextCommssionValue</td></tr> <tr> <td>D</td><td>StaticTextDownPaymentValue</td></tr> </tbody> </table>	Part	Name	A	StaticTextPriceSqft	B	StaticTextDaysOnMarket	C	StaticTextCommssionValue	D	StaticTextDownPaymentValue
Part	Name										
A	StaticTextPriceSqft										
B	StaticTextDaysOnMarket										
C	StaticTextCommssionValue										
D	StaticTextDownPaymentValue										
8	Change the text attribute of <code>IStaticText*</code> parts <b>A</b> , <b>B</b> , <b>C</b> , and <b>D</b> to <b>Not Available</b> and their respective limit to <b>7</b> , <b>3</b> , <b>7</b> , and <b>7</b> . Their contents will be calculated later by the nonvisual part, <code>AMarketingInfo</code> (see “ <code>AMarketingInfo</code> ” on page 226).										

**Table 13.** (Part 2 of 2) Building the Marketing Page

Step	Action												
9	<p>Add three IEntryField* parts (<b>E</b>, <b>F</b>, <b>G</b>) to cells (3, 5), (11, 5), and (18, 5) of the IMultiCellCanvas* part and set their part names and limits as follows:</p> <table><tr><th>Part</th><th>Name</th><th>Limit</th></tr><tr><td>E</td><td>EntryFieldPrice</td><td>7</td></tr><tr><td>F</td><td>EntryFieldCommissionRateSize</td><td>5</td></tr><tr><td>G</td><td>EntryFieldDownPaymentRate</td><td>5</td></tr></table>	Part	Name	Limit	E	EntryFieldPrice	7	F	EntryFieldCommissionRateSize	5	G	EntryFieldDownPaymentRate	5
Part	Name	Limit											
E	EntryFieldPrice	7											
F	EntryFieldCommissionRateSize	5											
G	EntryFieldDownPaymentRate	5											
10	<p>Add two IGroupBox* parts to group the commission rate and value and the down payment rate and value. The commission group box extends from cell (9, 2) to cell (14, 6), and the down payment group box extends from cell (16, 2) to cell (21, 6).</p>												
11	<p>Set the Tabbing and Depth Order of the different entry fields as follows (see “Setting the Tabbing and Depth Order” on page 157):</p> <table><tr><th>Group</th><th>Tab</th><th>Feature</th></tr><tr><td>X</td><td>X</td><td>EntryFieldPrice</td></tr><tr><td>X</td><td>X</td><td>EntryFieldCommissionRate</td></tr><tr><td>-</td><td>X</td><td>EntryFieldDownPaymentRate</td></tr></table> <p>Close the dialog box.</p>	Group	Tab	Feature	X	X	EntryFieldPrice	X	X	EntryFieldCommissionRate	-	X	EntryFieldDownPaymentRate
Group	Tab	Feature											
X	X	EntryFieldPrice											
X	X	EntryFieldCommissionRate											
-	X	EntryFieldDownPaymentRate											
<b>Note:</b> Reverse highlighted letters are keyed to Figure 62 on page 174.													

To reuse APropertyView in other parts, you must promote some of its features. Switch to the Part Interface Editor and select the Promote page. Promote the features listed in Table 14.

**Table 14.** (Part 1 of 2) Promoted Features of APropertyView

Promote Feature Name	Subpart Name	Feature Type	Promoted Feature
addressViewComboBoxStateText	AddressView	attribute	ComboBoxStateText
addressViewEntryFieldAreaText	AddressView	attribute	EntryFieldAreaText
addressViewEntryFieldCityText	AddressView	attribute	EntryFieldCityText
addressViewEntryFieldStreetText	AddressView	attribute	EntryFieldStreetText
addressViewEntryFieldZipCodeText	AddressView	attribute	EntryFieldZipCodeText

**Table 14.** (Part 2 of 2) Promoted Features of APropertyView

Promote Feature Name	Subpart Name	Feature Type	Promoted Feature
commissionRate	EntryFieldCommissionRate	attribute	valueAsDouble
daysOnMarket	StaticTextDaysOnMarket	attribute	text
downPaymentRate	EntryFieldDownPaymentRate	attribute	valueAsDouble
price	EntryFieldPrice	attribute	valueAsDouble
bathrooms	NumericSpinButtonBathrooms	attribute	value
bedrooms	NumericSpinButtonBedrooms	attribute	value
size	EntryFieldSize	attribute	valueAsDouble
propertyID	EntryFieldPropertyID	attribute	text
propertyIDReadOnly	EntryFieldPropertyID	action	disableDataUpdate
cooling	ComboBoxCooling	attribute	text
heating	ComboBoxHeating	attribute	text
videoFileName	EntryFieldVideo	attribute	text
<b>Note:</b> Notice that the numeric attributes are promoted as <i>valueAsDouble</i> .			

Notice that among the promoted features, you promote the **disableDataUpdate** action of EntryFieldPropertyID. This action sets the entry field writable attribute to **false** and prevents its contents from being updated. This precaution is necessary when using APropertyView within APropertyUpdateView (used to update the property information in the database) to prevent the user from editing the property identifier and changing the property key in the database.

Now you can save APropertyView. First, switch to the Class Interface Editor and fill in the *Code generation file* group box as follows:

- ☐ C++ header file (.hpp): **vrpprpv.hpp**
- ☐ C++ code file (.cpp): **vrpprpv.cpp**

Then save the part.

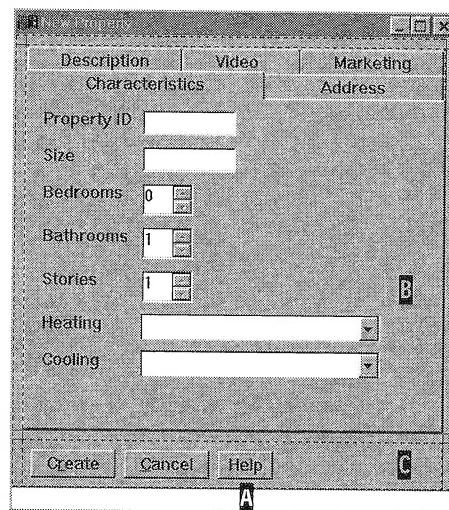
## APropertyCreateView

APropertyCreateView (Figure 63) is a composite part that consists of APropertyView and three push buttons:

- ☐ **Create**, for creating a new property in the database
- ☐ **Cancel**, for canceling the operation and closing the window
- ☐ **Help**, for accessing on-line help

The base class of this part is an IFrameWindow part. You tailor this main window, adding an information area, IInfoArea part, to its frame and changing the standard canvas to a multicell canvas. The information area is used to display help information related to some parts of the view. We explain the use of the information area with the fly-over help facility in “Adding Fly-over Help to a Control” on page 270.

The three push buttons are added to a set canvas.



**Figure 63.** APropertyCreateView

To build APropertyCreateView, follow the steps in Table 15.

<b>Table 15.</b> (Part 1 of 3) Constructing APropertyCreateView Part	
<b>Step</b>	<b>Action</b>
1	Start Visual Builder from the Property project (select <b>Project</b> → <b>Visual</b> in the menu bar).
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.

**Table 15.** (Part 2 of 3) Constructing APropertyCreateView Part




Step	Action												
3	<p>Fill in the entry fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Class name</td><td>APropertyCreateView</td></tr> <tr> <td>Description</td><td>View to create a property in the database</td></tr> <tr> <td>File name</td><td>VRPROP</td></tr> <tr> <td>Part type</td><td>visual part</td></tr> <tr> <td>Base class</td><td>IFrameWindow</td></tr> </table> <p>Click on the <b>Open</b> push button. An IFrameWindow* part is displayed on the free-form surface.</p>	Field	Value	Class name	APropertyCreateView	Description	View to create a property in the database	File name	VRPROP	Part type	visual part	Base class	IFrameWindow
Field	Value												
Class name	APropertyCreateView												
Description	View to create a property in the database												
File name	VRPROP												
Part type	visual part												
Base class	IFrameWindow												
4	Change the IFrameWindow* part title to <b>New Property</b> .												
5	Delete the ICanvas* part in the IFrameWindow*.												
6	Add an IMultiCellCanvas* part to the IFrameWindow*.												
7	<p>Open the settings notebook of the IMultiCellCanvas* part and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 4</li> <li><input type="checkbox"/> Number of columns: 3</li> <li><input type="checkbox"/> Expandable rows: Nos. 2, 3</li> <li><input type="checkbox"/> Expandable column: No. 2</li> </ul> <p>When you set Row 3 to expandable, the ISetCanvas* part that holds the three push buttons is kept in the bottom of the window when the view is resized.</p>												
8	Switch to the Color page and select the <b>Colors</b> radio button of the <i>Color selection</i> group box. Then select <b>paleGray</b> in the <i>Color values</i> drop-down list box. In this way, all controls dropped in the multicell canvas have a pale gray background. Close the settings notebook.												
9	Add an IInfoArea* part,  to the IFrameWindow* part. (The IInfoArea* part is located in the <i>Frame extensions</i> category).												
10	Add an APropertyView*  , to cell (2, 2) of the IMultiCellCanvas* part as shown in Figure 63 on page 178 ( <b>Option</b> → <b>Add parts</b> from the Composition Editor menu).												
11	Add an ISetCanvas* part,  , to cell (4, 2) of the IMultiCellCanvas* part as shown on Figure 63 on page 178.												



Table 15. (Part 3 of 3) Constructing APropertyCreateView Part														
Step	Action													
12	Add three IPushButton* parts to the ISetCanvas* part and set their names as follows:  <input type="checkbox"/> <b>PushButtonCreate</b> <input type="checkbox"/> <b>PushButtonCancel</b> <input type="checkbox"/> <b>PushButtonHelp</b>													
13	Change the <i>text</i> attribute of the three push buttons as follows:  <input type="checkbox"/> <b>~Create</b> for PushButtonCreate <input type="checkbox"/> <b>~Cancel</b> for PushButtonCancel <input type="checkbox"/> <b>~Help</b> for PushButtonHelp  Notice the use of ~ for the key accelerator.													
14	Open the settings notebook of PushButtonHelp and switch to the Styles page. Set the <i>help</i> radio button to <b>On</b> to turn this regular push button into a help push button. Set the <i>noPointerFocus</i> radio button to <b>On</b> to prevent the help push button from getting the input focus when the user clicks on it. In this way, the application can display help for the part that has the input focus when the user clicks the <i>help</i> push button. Close the notebook settings.													
15	Set the tabbing and depth order and a group setting for the three push buttons as follows (see “Setting the Tabbing and Depth Order” on page 157):  <table><tr><th>Group</th><th>Tab</th><th>Feature</th></tr><tr><td>X</td><td>X</td><td>PushButtonCreate</td></tr><tr><td>-</td><td>X</td><td>PushButtonCancel</td></tr><tr><td>-</td><td>X</td><td>PushButtonCreate</td></tr></table> Close the dialog box.		Group	Tab	Feature	X	X	PushButtonCreate	-	X	PushButtonCancel	-	X	PushButtonCreate
Group	Tab	Feature												
X	X	PushButtonCreate												
-	X	PushButtonCancel												
-	X	PushButtonCreate												
<b>Note:</b> Reverse highlighted letters are keyed to Figure 63 on page 178.														

Now you can save APropertyCreateView. First, switch to the Class Editor and fill in the *Code generation file* group box as follows:

- ☐ C++ header file (.hpp): **vrpcrtv.hpp**
- ☐ C++ code file (.cpp): **vrpcrtv.cpp**

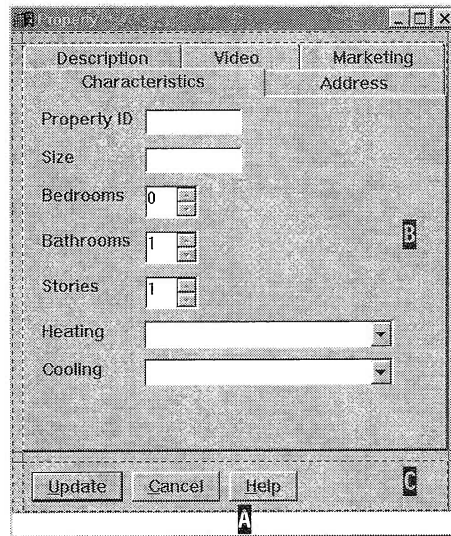
Then, save the part.

# APropertyUpdateView

APropertyUpdateView (Figure 64) is a composite part that consists of APropertyView and three push buttons:

- ❑ **Update**, for updating property information in the database
- ❑ **Cancel**, for canceling the operation and closing the window
- ❑ **Help**, for accessing on-line help

You build the APropertyUpdateView in the same way you built APropertyCreateView.






**Figure 64.** APropertyUpdateView

To build APropertyUpdateView, follow the steps in Table 16.

Table 16. (Part 1 of 3) Constructing APropertyUpdateView Part													
Step	Action												
1	Start Visual Builder from the Property project (select <b>Project</b> → <b>Visual</b> in the menu bar).												
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.												
3	<p>Fill in the entry fields:</p> <table border="1"> <thead> <tr> <th>Field</th><th>Value</th></tr> </thead> <tbody> <tr> <td>Class name</td><td>APropertyUpdateView</td></tr> <tr> <td>Description</td><td>View to update property information in the database</td></tr> <tr> <td>File name</td><td>VRPROP</td></tr> <tr> <td>Part type</td><td>visual part</td></tr> <tr> <td>Base class</td><td>IFrameWindow</td></tr> </tbody> </table> <p>Click on the <b>Open</b> push button. An IFrameWindow* part is displayed on the free-form surface.</p>	Field	Value	Class name	APropertyUpdateView	Description	View to update property information in the database	File name	VRPROP	Part type	visual part	Base class	IFrameWindow
Field	Value												
Class name	APropertyUpdateView												
Description	View to update property information in the database												
File name	VRPROP												
Part type	visual part												
Base class	IFrameWindow												

**Table 16.** (Part 2 of 3) Constructing APropertyUpdateView Part

Step	Action
4	Change the IFrameWindow* part title to <b>Property</b> .
5	Delete the ICanvas* part in the IFrameWindow* part.
6	Add an IMultiCellCanvas* part to the IFrameWindow*.
7	Open the settings notebook of the IMultiCellCanvas* part and configure it as follows: <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 4</li> <li><input type="checkbox"/> Number of columns: 3</li> <li><input type="checkbox"/> Expandable rows: Nos. 2, 3</li> <li><input type="checkbox"/> Expandable column: No. 2</li> </ul>
8	Switch to the Color page and select the <b>Colors</b> radio button of the <i>Color selection</i> group box. Then select <b>paleGray</b> in the <i>Color values</i> drop-down list box. In this way, all controls dropped in the multicell canvas have a pale gray background. Close the settings notebook.
9	Add an IInfoArea* part,  , to the IFrameWindow* part.
10	Add an APropertyView* part,  , to cell (2, 2) of the IMultiCellCanvas* part as shown in Figure 64 on page 181 ( <b>Option</b> → <b>Add parts</b> from the Composition Editor menu).
11	Add an ISetCanvas* part,  , to cell (4, 2) of the IMultiCellCanvas* part as shown on Figure 64 on page 181.
12	Add three IPushButton* parts to the ISetCanvas* part and set their names as follows: <ul style="list-style-type: none"> <li><input type="checkbox"/> <b>PushButtonUpdate</b></li> <li><input type="checkbox"/> <b>PushButtonCancel</b></li> <li><input type="checkbox"/> <b>PushButtonHelp</b></li> </ul>
13	Change the <i>text</i> attribute of the three push buttons as follows: <ul style="list-style-type: none"> <li><input type="checkbox"/> <b>~Update</b> for PushButtonUpdate</li> <li><input type="checkbox"/> <b>~Cancel</b> for PushButtonCancel</li> <li><input type="checkbox"/> <b>~Help</b> for PushButtonHelp</li> </ul> Notice the use of ~ for the key accelerator.
14	Open the settings notebook of PushButtonHelp and switch to the Styles page. Set the <i>help</i> radio button to <b>On</b> . Set the <i>noPointerFocus</i> radio button to <b>On</b> . Close the settings notebook.

**Table 16.** (Part 3 of 3) Constructing APropertyUpdateView Part

Step	Action												
15	<p>Set the Tabbing and Depth Order as follows (see “Setting the Tabbing and Depth Order” on page 157):</p> <table><tr><th>Group</th><th>Tab</th><th>Feature</th></tr><tr><td>X</td><td>X</td><td>PushButtonUpdate</td></tr><tr><td>-</td><td>X</td><td>PushButtonCancel</td></tr><tr><td>-</td><td>X</td><td>PushButtonCreate</td></tr></table> <p>Close the dialog box.</p>	Group	Tab	Feature	X	X	PushButtonUpdate	-	X	PushButtonCancel	-	X	PushButtonCreate
Group	Tab	Feature											
X	X	PushButtonUpdate											
-	X	PushButtonCancel											
-	X	PushButtonCreate											
<b>Note:</b> Reverse highlighted letters are keyed to Figure 64 on page 181.													

Now you can save APropertyUpdateView. First, switch to the Class Editor and fill in the *Code generation file* group box as follows:

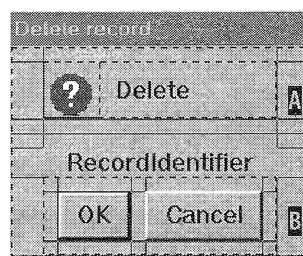
- ☐ C++ header file (.hpp): **vrpupdv.hpp**
- ☐ C++ code file (.cpp): **vrpupdv.cpp**

Then, save the part.

APropertyView and APropertyUpdateView look quite similar, but they differ in their push button labels. Reusing APropertyView saves a lot of time and brings consistency to the whole application.

## ADeleteDialogView

ADeleteDialogView is a simple visual part (Figure 65) that is used throughout the application. It provides a way of warning users when they go to delete a record in the database.

**Figure 65.** ADeleteDialogView

Because you reuse this part across different subsystems, you do not embed any Data Access Builder parts. Instead, you promote two features:

- ❑ The `buttonClickEvent` of the *OK* push button. The part that reuses `ADeleteDialogView` needs this promoted feature to know when the button has been clicked and to perform the appropriate action.
- ❑ The *text* attribute of the `textOfRecord` static text control. This promoted feature is used to display, in the dialog box, the identifier of the record to be deleted (see Table 17 on page 185 and Figure 65 on page 183).

`ADeleteDialogView` is used as a dialog window. It is not resizable, but you still have to use the `IMultiCellCanvas*` part to make this view portable across different screen resolutions and different font settings.

Even though this view is not resizable, you must set some rows and columns of the `IMultiCellCanvas*` part to be expandable. In effect, when running the application under different screen resolutions, this view is resized to fit the new resolution and the canvas may be a bit distorted. In setting the outermost columns and rows to be expandable, you minimize the distortion.

**Tip!**



When using a multicell canvas for your view, you can directly observe the effect of resizing the view in the Composition Editor and anticipate the appearance of your view in different resolutions.

Notice that you use an `IIconControl` part to display an icon in the dialog window. The icon displayed is set by updating the *DLL name* and *resource ID* fields in the `IIconControl` General settings page. Be aware that an icon is not resized on either a set canvas or a multicell canvas when the canvas is resized.

In our sample application, you use the `REALICON.DLL` provided with this book in the CD ROM. It contains predefined icons that you can use for your own needs. In “Resource DLL Creation” on page 129, we show you how to build such a resource Dynamic Link Library using Project Smarts.

To ensure that your application can access this DLL, put it in a directory that your `LIBPATH` accesses (check the `LIBPATH` environment variable in your `CONFIG.SYS` file or in your System Variables).

To build ADeleteDialogView, follow the steps in Table 17.

<b>Table 17.</b> (Part 1 of 3) Building ADeleteDialogView													
<b>Step</b>	<b>Action</b>												
1	Start Visual Builder from the Common project (select <b>Project</b> → <b>Visual</b> in the menu bar).												
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.												
3	<p>Fill in the entry fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Class name</td><td>ADeleteDialogView</td></tr> <tr> <td>Description</td><td>General purpose delete dialog view</td></tr> <tr> <td>File name</td><td>VRCOMM</td></tr> <tr> <td>Part type</td><td>visual part</td></tr> <tr> <td>Base class</td><td>IFrameWindow</td></tr> </table> <p>and click on the <b>Open</b> push button. An IFrameWindow* part is displayed on the free-form surface.</p>	Field	Value	Class name	ADeleteDialogView	Description	General purpose delete dialog view	File name	VRCOMM	Part type	visual part	Base class	IFrameWindow
Field	Value												
Class name	ADeleteDialogView												
Description	General purpose delete dialog view												
File name	VRCOMM												
Part type	visual part												
Base class	IFrameWindow												
4	Change the IFrameWindow* part title to <b>Delete Record</b> .												
5	<p>Open the settings notebook of the IFrameWindow* part and change the style setting as follows:</p> <table> <tr> <th>Setting</th><th>Value</th></tr> <tr> <td>dialogBorder</td><td>On</td></tr> <tr> <td>maximizeButton</td><td>Off</td></tr> <tr> <td>minimizeButton</td><td>Off</td></tr> <tr> <td>sizingBorder</td><td>Off</td></tr> <tr> <td>systemMenu</td><td>Off</td></tr> </table> <p>Close the settings notebook. The window turns into a nonresizable dialog box.</p>	Setting	Value	dialogBorder	On	maximizeButton	Off	minimizeButton	Off	sizingBorder	Off	systemMenu	Off
Setting	Value												
dialogBorder	On												
maximizeButton	Off												
minimizeButton	Off												
sizingBorder	Off												
systemMenu	Off												
6	Delete the ICanvas* part from the IFrameWindow* part.												
7	Add an IMultiCellCanvas* part to the IFrameWindow* part.												
8	<p>Open the settings notebook of the IMultiCellCanvas* part and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 7</li> <li><input type="checkbox"/> Number of columns: 3</li> <li><input type="checkbox"/> Expandable rows: Nos. 3, 5</li> <li><input type="checkbox"/> Expandable columns: Nos. 1, 3</li> </ul>												
9	Switch to the Color page and select the <b>Colors</b> radio button of the <i>Color selection</i> group box. Then select <b>paleGray</b> in the <i>Color values</i> drop-down list box. Close the settings notebook.												

**Table 17.** (Part 2 of 3) Building ADeleteDialogView

Step	Action										
10	Add an ISetCanvas* part, <b>A</b> , to cell (2, 2) of the IMultiCellCanvas* part. This set canvas will contain an icon and a static text control lined up horizontally.										
11	Add an IIconControl* to the ISetCanvas* part. (The IIconControl* part is located in the <i>Data entry</i> category.)										
12	Open the settings notebook of the IIconControl* part. Set the DLL name to <b>realicon</b> and the resource ID to <b>502</b> . Close the settings notebook.										
13	Add an IStaticText* part to the ISetCanvas* part and change its label to <b>Delete</b> .										
14	<p>Open the settings notebook of the ISetCanvas* part, set its alignment to center (select the middle radio button in the <i>Alignment</i> group box), and adjust its margin and pad dimensions as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Margin Width</td><td>0</td></tr> <tr> <td>Margin Height</td><td>0</td></tr> <tr> <td>Pad Width</td><td>10</td></tr> <tr> <td>Pad Height</td><td>0</td></tr> </table> <p>Close the settings notebook.</p>	Field	Value	Margin Width	0	Margin Height	0	Pad Width	10	Pad Height	0
Field	Value										
Margin Width	0										
Margin Height	0										
Pad Width	10										
Pad Height	0										
15	Add an IStaticText* part to cells (5, 2) of the IMultiCellCanvas* part and change its label to <b>RecordIdentifier</b> .										
16	Open the settings notebook of the RecordIdentifier IStaticText* part and change the part's name to <b>RecordID</b> . On the same page, set the static text alignment to center (select the middle radio button of the <i>Alignment</i> group box). Close the settings notebook.										
17	Promote the <b>text</b> attribute of the RecordID part.										
18	<p>Add an IMultiCellCanvas* part, <b>B</b>, to cell (7, 2) of the IMultiCellCanvas* part and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 3</li> <li><input type="checkbox"/> Number of columns: 5</li> <li><input type="checkbox"/> Expandable rows: none</li> <li><input type="checkbox"/> Expandable columns: Nos. 1, 3, 5</li> </ul>										
19	Add two IPushButton* parts to cells (2, 2) and (2, 4) of the IMultiCellCanvas* part, <b>B</b> , and change their labels as shown in Figure 65 on page 183.										

**Table 17.** (Part 3 of 3) Building ADeleteDialogView

Step	Action									
20	Change the push button names to <b>PushButtonOK</b> and <b>PushButtonCancel</b> .									
21	Promote the <b>buttonClickEvent</b> event of <i>PushbuttonOK</i> .									
22	Set <b>PushbuttonOK</b> as the default push button (set defaultButton to <b>On</b> in the Styles page of the settings notebook).									
23	Set the Tabbing and Depth Order of the push buttons as follows (see “Setting the Tabbing and Depth Order” on page 157): <table><tr><th>Group</th><th>Tab</th><th>Feature</th></tr><tr><td>-</td><td>X</td><td>PushButtonOK</td></tr><tr><td>-</td><td>X</td><td>PushButtonCancel</td></tr></table>	Group	Tab	Feature	-	X	PushButtonOK	-	X	PushButtonCancel
Group	Tab	Feature								
-	X	PushButtonOK								
-	X	PushButtonCancel								
<b>Note:</b> Reverse highlighted letters are keyed to Figure 65 on page 183.										

Now you can save ADeleteDialogView. First, switch to the Class Editor and fill in the *Code generation file* group box as follows:

- ☐ C++ header file (.hpp): **vrcdelv.hpp**
- ☐ C++ code file (.cpp): **vrcdelv.cpp**

Then, save the part.

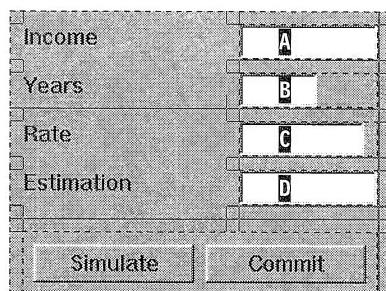
## ASimulMortgageView

ASimulMortgageView provides the user with a simple dialog box to enter the following information (Figure 66):

- ☐ Annual income
- ☐ Number of years
- ☐ Interest rate

These items are used by the AMortgageCalculator nonvisual part (which you develop in Chapter 8, “Creating Nonvisual Parts,” on page 225) to estimate the highest mortgage the buyer can contract and thereby the houses he can afford to buy. ASimulMortgageView and AMortgageCalculator fulfill the requirement “Show affordable properties” described in “Requirement Specifications” on page 62.





**Figure 66.** ASimulMortgageView

Because you reuse ASimulMortgageView (see Chapter 13, “More about CDF...,” on page 453) to develop an OLE server, you provide this visual part with a separate frame window. (When building an OLE component, the frame window is provided by the Compound Document Framework throughout the GUI bundle class.) ASimulMortgageFrameView is used for this purpose. ASimulMortgageView inherits from IMultiCellCanvas and is embedded in ASimulMortgageFrameView. To access the estimation entry field contents and the click events of the two push buttons from ASimulMortgageFrameView, you must also promote several attributes and events (see “Promoting a Part Feature” on page 158 for more information).

ASimulMortgageView is simple. The only peculiarity is two event handlers, which you attach to the first three entry fields to force the user to enter numerical or numerical-with-decimal-point information (the fourth entry field does not need one since it has a *read only* style). We explain how to construct your own event handler in “Event Handler” on page 237.

To build ASimulMortgageView, follow the steps in Table 18.

<b>Table 18.</b> (Part 1 of 3) Building ASimulMortgageView	
<b>Step</b>	<b>Action</b>
1	Start Visual Builder from the Common project (select <b>Project</b> → <b>Visual</b> in the menu bar).
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.

**Table 18.** (Part 2 of 3) Building ASimulMortgageView

Step	Action															
3	<p>Fill in the entry fields as follows:</p> <table><tr><th>Field</th><th>Value</th></tr><tr><td>Class name</td><td>ASimulMortgageView</td></tr><tr><td>Description</td><td>Simple mortgage simulation dialog view</td></tr><tr><td>File name</td><td>VRSERV</td></tr><tr><td>Part type</td><td>visual part</td></tr><tr><td>Base class</td><td>IMultiCellCanvas</td></tr></table> <p>and click on the <b>Open</b> push button. An IMultiCellCanvas* part is displayed on the free-form surface.</p>	Field	Value	Class name	ASimulMortgageView	Description	Simple mortgage simulation dialog view	File name	VRSERV	Part type	visual part	Base class	IMultiCellCanvas			
Field	Value															
Class name	ASimulMortgageView															
Description	Simple mortgage simulation dialog view															
File name	VRSERV															
Part type	visual part															
Base class	IMultiCellCanvas															
4	<p>Open the settings notebook of the IMultiCellCanvas* part and configure it as follows:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> Number of rows: 11</li><li><input type="checkbox"/> Number of columns: 5</li><li><input type="checkbox"/> Expandable rows: Nos. 1, 10</li><li><input type="checkbox"/> Expandable columns: Nos. 1, 5</li></ul>															
5	<p>Switch to the Color page and select the <b>Colors</b> radio button of the <i>Color selection</i> group box. Then select <b>paleGray</b> in the <i>Color values</i> drop-down list box. Close the settings notebook.</p>															
6	<p>Add four IStaticText* to cells (2, 2), (4, 2), (6, 2) and (8, 2) of the IMultiCellCanvas* part. Change their text attribute to <i>Income</i>, <i>Years</i>, <i>Rate</i>, and <i>Estimation</i> as shown in Figure 66.</p>															
7	<p>Add four ISetCanvas* to cells (2, 4), (4, 4), (6, 4) and (8, 4) of the IMultiCellCanvas* part and extends the last ISetCanvas* part to Column 3 (hold <b>ALT</b> key while dragging the ISetCanvas handles).</p>															
8	<p>Open the settings notebook of the ISetCanvas* part located in cells (2, 4), (4, 4), (6,4), and adjust the margins as follows:</p> <table><tr><th>Field</th><th>Value</th></tr><tr><td>Margin Width</td><td>0</td></tr><tr><td>Margin Height</td><td>0</td></tr></table> <p>Close each settings notebook.</p>	Field	Value	Margin Width	0	Margin Height	0									
Field	Value															
Margin Width	0															
Margin Height	0															
9	<p>Add four IEntryField* parts (<b>A</b>, <b>B</b>, <b>C</b>, <b>D</b>) within each ISetCanvas* part and set their part names and limits as follows:</p> <table><tr><th>Part</th><th>Name</th><th>Limit</th></tr><tr><td>A</td><td>EntryFieldIncome</td><td>7</td></tr><tr><td>B</td><td>EntryFieldYears</td><td>2</td></tr><tr><td>C</td><td>EntryFieldRate</td><td>5</td></tr><tr><td>D</td><td>EntryFieldEstimation</td><td>7</td></tr></table>	Part	Name	Limit	A	EntryFieldIncome	7	B	EntryFieldYears	2	C	EntryFieldRate	5	D	EntryFieldEstimation	7
Part	Name	Limit														
A	EntryFieldIncome	7														
B	EntryFieldYears	2														
C	EntryFieldRate	5														
D	EntryFieldEstimation	7														

**Table 18.** (Part 3 of 3) Building ASimulMortgageView

Step	Action																					
10	Open the settings notebook of EntryFieldIncome and, on the Handlers page, add NumDecOnlyKbdHandler to the handler list box. Close the settings notebook.																					
11	Open the settings notebook of EntryFieldYears and, on the Handlers page, add NumOnlyKbdHandler to the handler list box. Close the settings notebook.																					
12	Open the settings notebook of EntryFieldRate and, on the Handlers page, add an NumDecOnlyKbdHandler to the handler list box. Close the settings notebook.																					
13	Open the settings notebook of EntryFieldEstimation and in the page Styles page, set the <b>readOnly</b> radio button to <b>On</b> . This prevents the user from editing the field.																					
14	Add two IPushButton* parts to cell to the last ISetCanvas* part, and change their labels as shown in Figure 66 on page 188.																					
15	Change the push button names to <b>PushButtonSimulate</b> and <b>PushButtonCommit</b> .																					
16	Promote the <i>text</i> attribute of <i>EntryFieldEstimation</i> .																					
17	Promote the <b>buttonClickEvent</b> event of <i>PushbuttonSimulate</i> and <i>PushButtonCommit</i> .																					
18	Set <b>PushbuttonSimulate</b> as the default push button (set defaultButton to <b>On</b> in the Styles page of the settings notebook).																					
19	<p>Set the Tabbing and Depth Order and the Group settings of the entry fields and push buttons as follows (see “Setting the Tabbing and Depth Order” on page 157):</p> <table><tr><th>Group</th><th>Tab</th><th>Feature</th></tr><tr><td>X</td><td>X</td><td>EntryFieldIncome</td></tr><tr><td>-</td><td>X</td><td>EntryFieldYears</td></tr><tr><td>-</td><td>X</td><td>EntryFieldRate</td></tr><tr><td>-</td><td>X</td><td>EntryFieldEstimation</td></tr><tr><td>X</td><td>X</td><td>PushButtonSimulate</td></tr><tr><td>-</td><td>X</td><td>PushButtonCancel</td></tr></table>	Group	Tab	Feature	X	X	EntryFieldIncome	-	X	EntryFieldYears	-	X	EntryFieldRate	-	X	EntryFieldEstimation	X	X	PushButtonSimulate	-	X	PushButtonCancel
Group	Tab	Feature																				
X	X	EntryFieldIncome																				
-	X	EntryFieldYears																				
-	X	EntryFieldRate																				
-	X	EntryFieldEstimation																				
X	X	PushButtonSimulate																				
-	X	PushButtonCancel																				
<b>Note:</b> Reverse highlighted letters are keyed to Figure 66 on page 188.																						

Now you can save ASimulMortgageView. First, switch to the Class Editor and fill in the *Code generation file* group box as follows:

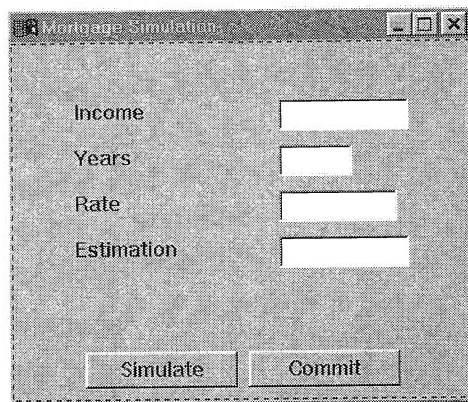
❑ C++ header file (.hpp): **vrssimv.hpp**

□ C++ code file (.cpp): `vrssimv.cpp`

Then, save the part.

## ASimulMortgageFrameView

ASimulMortgageFrameView provides ASimulMortgageView with a frame and its extension. This view inherits from IFrameWindow and ASimulMortgageView represents its client area (Figure 67).



**Figure 67.** ASimulMortgageFrameView

To build ASimulMortgageFrameView, follow the steps in Table 19.

Table 19. (Part 1 of 2) Building ASimulMortgageFrameView													
Step	Action												
1	Start Visual Builder from the Common project (select <b>Project</b> → <b>Visual</b> in the menu bar).												
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.												
3	<p>Fill in the entry fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Class name</td><td>ASimulMortgageFrameView</td></tr> <tr> <td>Description</td><td>Frame view for ASimulMortgageView</td></tr> <tr> <td>File name</td><td>VRSERV</td></tr> <tr> <td>Part type</td><td>visual part</td></tr> <tr> <td>Base class</td><td>IFrameWindow</td></tr> </table> <p>and click on the <b>Open</b> push button. An IFrameWindow* part is displayed on the free-form surface.</p>	Field	Value	Class name	ASimulMortgageFrameView	Description	Frame view for ASimulMortgageView	File name	VRSERV	Part type	visual part	Base class	IFrameWindow
Field	Value												
Class name	ASimulMortgageFrameView												
Description	Frame view for ASimulMortgageView												
File name	VRSERV												
Part type	visual part												
Base class	IFrameWindow												

**Table 19.** (Part 2 of 2) Building ASimulMortgageFrameView

Step	Action
4	Change the IFrameWindow* part title to <b>Mortgage Simulation</b> .
5	Delete the ICanvas* part from the IFrameWindow* part.
10	Add the ASimulMortgageView* part inside the IFrameWindow* part by using the <b>Options</b> → <b>Add part...</b> option from the Visual Builder menu bar.

Now you can save ASimulMortgageFrameView. First, switch to the Class Editor and fill in the *Code generation file* group box as follows:

- ☐ C++ header file (.hpp): **vrssimfv.hpp**
- ☐ C++ code file (.cpp): **vrssimfv.cpp**

Then, save the part.

## APropertySearchResultView

APropertySearchResultView (Figure 68) displays a list of properties that match the buyer's criteria. To display these properties in tabular form, use an IVBContainerControl part, which represents a container control. The IVBContainerControl part can be compared to the CListCtrl class of MFC which encapsulates the functionality of a "list view control."

### Using a Container

An IVBContainerControl part is a control used to display nonvisual interface objects. As a container, it shows different views of the objects it holds:

**Text and flowed text view:** The objects are represented as text in single (text view) or multiple (flowed text view) columns.

**Name and flowed name view:** The objects are represented as small icons followed by text in single (name view) or multiple (flowed name view) columns.

**Icon view:** The objects are represented as icons.

**Tree view:** The objects are represented hierarchically.

**Details view:** The objects are represented as a table, with one row for each object and a column for each object attribute. This is the view you use to display the property list (Figure 68).



### Portability



As with the INotebook control, you can use a container style that is native to the system or a style that is compatible with the OS/2 Presentation Manager (PM) container. The styles of these containers differ considerably in their details view:

- The PM-compatible container can automatically size columns according to the data it shows. It also allows different icons in any number of columns. The user can select an object by clicking with the mouse on the object row in any column.
- The Windows native container does not automatically size columns for data, but the user can dynamically resize container columns by dragging column separators. The container shows only the object icon specified for the container. The icon is always defined in Column 1. The object text specified for the container is always defined in Column 2. No column separator appears between these two columns, so they appear as a single object identification column. The user can select an object with the mouse by clicking on the object row only in the object identification column.

For our sample application, you will use the details view of the Prop\_ad\_log Data Access Builder part. The Prop\_ad\_log attributes are displayed in column controls, which are added to the container. The attributes displayed are: identifier, state, city, and area. In addition, you tailor the container by adding an icon of a property.

PROPERTY				
Id	St.	City	Area	
 Obj...	Stri...	String 1	String 1	
 Obj...	Stri...	String 2	String 2	

**Figure 68.** APropertySearchResultView

APropertySearchResultView is built from an IFrameWindow part. An IMultiCellCanvas part is used as the client area in place of the standard ICanvas part.

You build the view in two steps: First you tailor a container to suit your needs, then you add five columns to the container and tailor each of them to display the necessary information (an icon of the property, the property status, the city, the state, and the area).

To build the container, follow the steps in Table 20.

<b>Table 20.</b> (Part 1 of 2) Building APropertySearchResultView: Building a Container													
<b>Step</b>	<b>Action</b>												
1	Start Visual Builder from the Property project (select <b>Project</b> → <b>Visual</b> in the menu bar).												
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.												
3	Fill in the entry fields as follows: <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Class name</td><td>APropertySearchResultView</td></tr> <tr> <td>Description</td><td>View of a property list</td></tr> <tr> <td>File name</td><td>VRPROP</td></tr> <tr> <td>Part type</td><td>visual part</td></tr> <tr> <td>Base class</td><td>IFrameWindow</td></tr> </table>	Field	Value	Class name	APropertySearchResultView	Description	View of a property list	File name	VRPROP	Part type	visual part	Base class	IFrameWindow
Field	Value												
Class name	APropertySearchResultView												
Description	View of a property list												
File name	VRPROP												
Part type	visual part												
Base class	IFrameWindow												
4	Change the IFrameWindow* part title to <b>Property Search Result</b> .												
5	Delete the ICanvas* part from the IFrameWindow* part.												
6	Add an IMultiCellCanvas* part to the IFrameWindow* part.												
7	Open the settings notebook of the IMultiCellCanvas* part and configure it as follows: <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of columns: 1</li> <li><input type="checkbox"/> Number of rows: 1</li> <li><input type="checkbox"/> Expandable columns: No. 1</li> <li><input type="checkbox"/> Expandable rows: No. 1</li> </ul> <p>You do not have to add extra rows and columns in the IMultiCellCanvas* part, because the container can fill the entire window client area.</p>												
8	Add an IVBContainerControl* part to the IMultiCellCanvas* part. (The IVBContainerControl* part is located in the <i>Lists</i> category.)												

**Table 20.** (Part 2 of 2) Building APropertySearchResultView: Building a Container

Step	Action																				
9	<p>Open the settings notebook of the IVBContainerControl* part and set the following values (Figure 69 on page 197):</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Subpart name</td><td>PropertyContainer</td></tr> <tr> <td>Title</td><td>PROPERTY</td></tr> <tr> <td>Show title</td><td>selected</td></tr> <tr> <td>Show title separator</td><td>selected</td></tr> <tr> <td>Title alignment</td><td>centered</td></tr> <tr> <td>View type</td><td>showDetailsView</td></tr> <tr> <td>Item type</td><td>Prop_ad_log*</td></tr> <tr> <td>Text</td><td>property_idAsString</td></tr> <tr> <td>Icon</td><td>#IDynamicLinkLibrary("realicon").loadIcon(510)</td></tr> </table> <p>Notice that the item type field is filled in with the type of the Data Access Builder part that maps the PROP_AD_LOG table. The container is tailored by displaying a “house” icon. This icon will be displayed in the icon column that you add later on.</p>	Field	Value	Subpart name	PropertyContainer	Title	PROPERTY	Show title	selected	Show title separator	selected	Title alignment	centered	View type	showDetailsView	Item type	Prop_ad_log*	Text	property_idAsString	Icon	#IDynamicLinkLibrary("realicon").loadIcon(510)
Field	Value																				
Subpart name	PropertyContainer																				
Title	PROPERTY																				
Show title	selected																				
Show title separator	selected																				
Title alignment	centered																				
View type	showDetailsView																				
Item type	Prop_ad_log*																				
Text	property_idAsString																				
Icon	#IDynamicLinkLibrary("realicon").loadIcon(510)																				
10	Switch to the Styles page and make sure that the <i>pmCompatible</i> style is set to <b>Off</b> (or Default).																				

In the same way you customize your notebook control, make sure that the **pmCompatible** style is set the way you want:

- ☐ If you select the **On** radio button, you have access only to functions that are common to OS/2 and Windows. The compiled container looks like a CUA container, even in Windows.
- ☐ If you select the **Off** radio button, you have access to all functions in the native container. This functions differs between OS/2 and Windows. In OS/2, the compiled container looks like a CUA container. In Windows, the compiled container looks like a native Windows container.



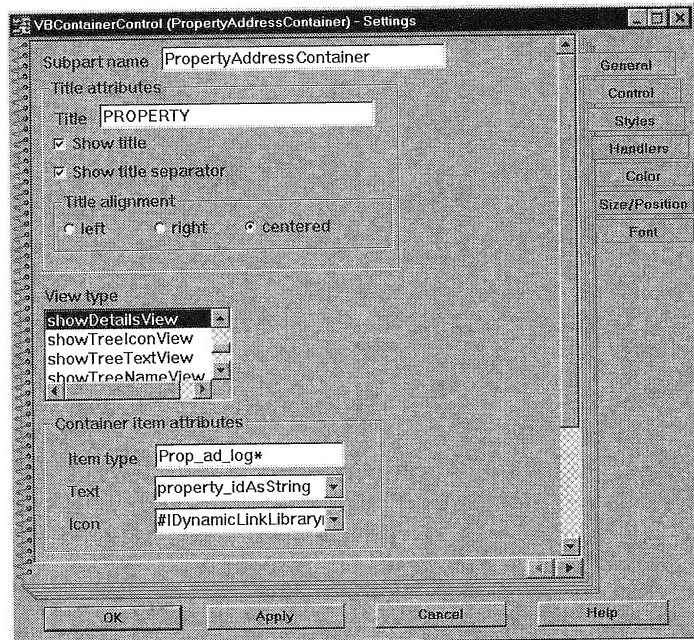
**Warning**

If you use the details view of a Windows style container, make sure that you define the first column with an icon and a label, otherwise you will not be able to select any object displayed in the container. To define the first column:

- Open the settings notebook of your container.
- In the *Text* drop-down list box of the Container item attributes, select a text to be displayed in the first column.
- Optionally associate an icon to the container in the *Icon* drop-down list box.
- Add one column to your container.
- Open the notebook settings of your first column and make sure the **Use Text attribute set in the Container** is selected in the *Column definition* group box.

If you provide your own container icon, you can add one more column and set its column definition to **Use Text attribute set in the Container**. The two columns will be merged as one. A default icon is provided for your container if you do not provide any.

When you enter the type of the object container, you must be aware of the object type that the object provider provides. In the sample application, the container is filled with the objects in the Data Access Builder Prop\_ad\_logManager part. Here, Prop\_ad\_logManager constitutes the object provider. As mentioned in Chapter 6, “Mapping Relational Tables Using Data Access Builder,” on page 131, we know that Prop\_ad\_logManager contains an attribute, *items*, of type IVSequence<Prop\_ad\_log\*>\*. This attribute will be connected by an attribute-to-attribute connection to the same attribute of the container. Thus, the container will hold a sequence of objects of Prop\_ad\_log\* type (Figure 69). Notice also that as explained in the *Warning* box above, the property identifier would appear as text in the container’s icon view. And the default icon for the property object would be the resource icon 510 in REALICON.DLL (this icon represents a little house).



**Figure 69.** Container General Settings Page

Once the container is set up, you must add container columns to display the information.

## Adding Columns to a Container

The detail view requires that you add a container column to the container for each object attribute to be displayed. A container column is represented by the `IContainerColumn*` part.

You add an `IContainerColumn*` part to your container by dragging it from the parts palette and dropping it on the container. Then, you edit its settings to reflect the information you want to display (Figure 70).

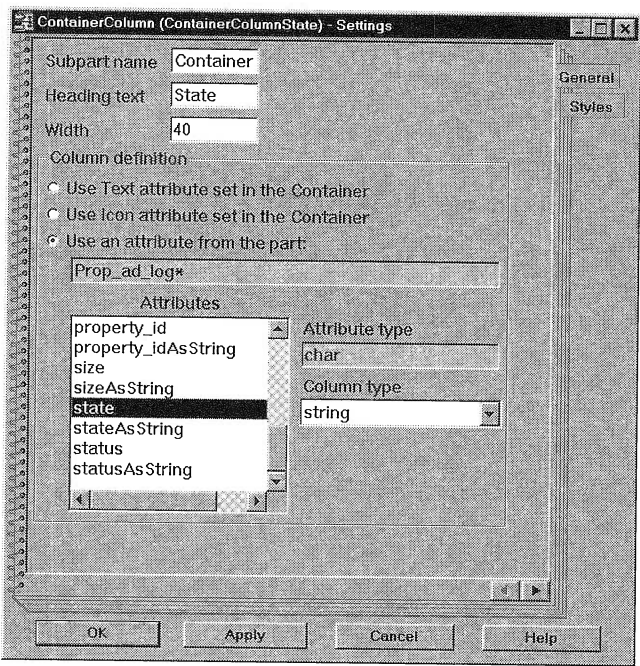


Figure 70. Container Column General Settings Page

To add the columns to your container, follow the steps in Table 21.

Table 21. (Part 1 of 3) Building APropertySearchResultView: Adding Container Columns											
Step	Action										
1	Add five IContainerColumn* parts in the IContainerControl* part. (The IContainerColumn* part is located in the Lists category.)										
2	<div>Open the settings notebook of the first IContainerColumn* part and change the fields as follows:<table><tr><th>Field</th><th>Value</th></tr><tr><td>Subpart name</td><td>ContainerColumnIcon</td></tr><tr><td>Heading text</td><td>Icon</td></tr><tr><td>Width</td><td>0</td></tr><tr><td>Use Icon attribute set in the container</td><td>selected</td></tr></table></div> <div>Close the settings notebook. The resource identifier set in the IContainerControl* part is used to display an icon in this IContainerColumn* part. Switch to the Styles page and set the <i>verticalSeparator</i> and <i>horizontalSeparator</i> to <b>On</b>. Close the settings notebook.</div>	Field	Value	Subpart name	ContainerColumnIcon	Heading text	Icon	Width	0	Use Icon attribute set in the container	selected
Field	Value										
Subpart name	ContainerColumnIcon										
Heading text	Icon										
Width	0										
Use Icon attribute set in the container	selected										

**Table 21.** (Part 2 of 3) Building APropertySearchResultView: Adding Container Columns

Step	Action												
3	<p>Open the settings notebook of the second IContainerColumn* part and change the fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Subpart name</td><td>ContainerColumnId</td></tr> <tr> <td>Heading text</td><td>Id</td></tr> <tr> <td>Width</td><td>75</td></tr> <tr> <td>Use Text attribute set in the container</td><td>selected</td></tr> <tr> <td>Attributes</td><td>property_id</td></tr> </table> <p>Close the settings notebook. The property identifier is displayed in this column. Switch to the Styles page and set the <i>horizontalSeparator</i> and <i>verticalSeparator</i> to <b>On</b>. Close the settings notebook.</p>	Field	Value	Subpart name	ContainerColumnId	Heading text	Id	Width	75	Use Text attribute set in the container	selected	Attributes	property_id
Field	Value												
Subpart name	ContainerColumnId												
Heading text	Id												
Width	75												
Use Text attribute set in the container	selected												
Attributes	property_id												
4	<p>Open the settings notebook of the third IContainerColumn* part and change the fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Subpart name</td><td>ContainerColumnState</td></tr> <tr> <td>Heading text</td><td>State</td></tr> <tr> <td>Width</td><td>40</td></tr> <tr> <td>Use an attribute from the part</td><td>selected</td></tr> <tr> <td>Attributes</td><td>state</td></tr> </table> <p>Close the settings notebook. The property status is displayed in this column. Then, switch to the Styles page and set the <i>horizontalSeparator</i> and <i>verticalSeparator</i> to <b>On</b>. Close the settings notebook.</p>	Field	Value	Subpart name	ContainerColumnState	Heading text	State	Width	40	Use an attribute from the part	selected	Attributes	state
Field	Value												
Subpart name	ContainerColumnState												
Heading text	State												
Width	40												
Use an attribute from the part	selected												
Attributes	state												
5	<p>Open the settings notebook of the fourth IContainerColumn* part and change the fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Subpart name</td><td>ContainerColumnCity</td></tr> <tr> <td>Heading text</td><td>City</td></tr> <tr> <td>Width</td><td>160</td></tr> <tr> <td>Use an attribute from the part</td><td>selected</td></tr> <tr> <td>Attributes</td><td>city</td></tr> </table> <p>Close the settings notebook. The property city is displayed in this column. Then, switch to the Styles page and set the <i>horizontalSeparator</i> and <i>verticalSeparator</i> to <b>On</b>. Close the settings notebook.</p>	Field	Value	Subpart name	ContainerColumnCity	Heading text	City	Width	160	Use an attribute from the part	selected	Attributes	city
Field	Value												
Subpart name	ContainerColumnCity												
Heading text	City												
Width	160												
Use an attribute from the part	selected												
Attributes	city												

Table 21. (Part 3 of 3) Building APropertySearchResultView: Adding Container Columns													
Step	Action												
6	<p>Open the settings notebook of the last IContainerColumn* part and change the fields as follows:</p> <table><tr><th>Field</th><th>Value</th></tr><tr><td>Subpart name</td><td>ContainerColumnArea</td></tr><tr><td>Heading text</td><td>Area</td></tr><tr><td>Width</td><td>160</td></tr><tr><td>Use an attribute from the part</td><td>selected</td></tr><tr><td>Attributes</td><td>area</td></tr></table> <p>Close the settings notebook. The property area is displayed in this column. Switch to the Styles page and set the horizontalSeparator and verticalSeparator to <b>On</b>. Close the settings notebook.</p>	Field	Value	Subpart name	ContainerColumnArea	Heading text	Area	Width	160	Use an attribute from the part	selected	Attributes	area
Field	Value												
Subpart name	ContainerColumnArea												
Heading text	Area												
Width	160												
Use an attribute from the part	selected												
Attributes	area												

Now you can save APropertySearchResultView. First, switch to the Class Editor and fill in the *Code generation file* group box as follows:

- ☐ C++ header file (.hpp): **vrpsrrsv.hpp**
- ☐ C++ code file (.cpp): **vrpsrrsv.cpp**

Then, save the part.

## APropertySearchParameterView

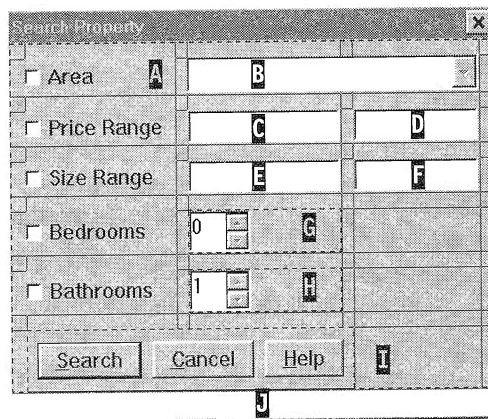
In the Visual Realty application, the user can search properties according to the buyer’s preferences. As stated in “Requirement Specifications” on page 62, different criteria are taken into account: area, price range, size range, number of bedrooms, and number of bathrooms. The user may choose to search properties using some or all of these criteria.

One way of designing a visual part that enables the user to construct a search-properties clause is to build a part that holds as many input controls as search criteria and use an ICheckBox part to select or deselect each criterion.

### Using Check Box Control

An ICheckBox part is a square box with text that represents the settings choice (Figure 71). A mark in the check box indicates that the choice is selected. In our case, you use the ICheckBox part instead of the IRadioButton part because the choices are not mutually exclusive.

For example, if the user wants to search all properties that range in size from 500 to 2500 square feet, the user first selects the **Size** check-box and then enters the size range.



**Figure 71.** APropertySearchParameterView

The part is built from an IFrameWindow part. An IMultiCellCanvas\* part is used in the client area to display the parts when the window is resized. Several ICheckBox\* and IEntryField\* parts are dropped on the IMultiCellCanvas. An ISetCanvas\* holds three IPushButton\* parts.

The list of areas is displayed in a collection combo-box control: ICollectionViewComboBox\*.

## Using Collection Combination-Box Control

An ICollectionViewComboBox part is a control that combines a selection list and an entry field for collection object choices. This selection list displays the records of the LIST\_AREA table. You will use the List\_areaManager Data Access Builder part to fill in the list. As with an IVBContainerControl part, the ICollectionViewComboBox part must be set up to display objects of a specific type. The type is entered in the *Item type* entry field of the General page in the settings notebook of the ICollectionViewComboBox part. The type must be a type pointer of the part that is displayed in the ICollectionViewComboBox part. In our case, you must set the type to **List\_area\***.

The ICollectionViewComboBox part combines the behavior of an IEntryField part with an ICollectionViewListBox part. It behaves like the ICollectionViewListBox part (ICollectionViewComboBox and ICollectionViewListBox parts are also called *collection* list box). When an

object is added to an `ICollectionViewListBox` part or an `ICollectionViewComboBox` part, the display of its contents is ruled by its method: `IString asString()`. This method returns an `IString`, which can be the concatenation of several object attributes.

For example, the parts that Data Access Builder generates have an `asString` method that returns the concatenation of all of their attributes, after conversion if necessary, separated by a dot. This is why, in the first example that you built in the Section “Using Data Access Builder Parts with Visual Builder” on page 140, the `ICollectionViewListBox` part displays each property part as the concatenation of the property identifier, property size, number of stories, number of bathrooms, number of bedrooms, type of cooling, type of heating, and description text. In our example, the only information displayed by the `asString` method of the `List_area` part is the property area, because it is the only attribute of the corresponding relational view. When the part does not have an `asString` method, *IVBase Object* is displayed instead.

You can tailor the information displayed in a collection list box in two ways:

- ❑ By overriding the `asString` method of the part that must be displayed in the collection list box.
- ❑ By providing a string generator class that produces the text you want for the collection class. You can create a string generator from scratch or by tailoring the behavior of an existing string generator class.

In the following section we choose to create a string generator class and associate it with the collection list box. In “Enhancing the Data Access in the Application” on page 413, we detail how to override the `asString` method of a Data Access Builder Generated part and how to override the behavior of an existing string generator class by overriding its `forDisplay` method.

### ***Creating a String Generator Class***

You can override the `ICollectionViewListBox` or the `ICollectionViewComboBox` string generator that customizes the part information to be displayed. A string generator, `IStringGenerator<Element>`, is a template class that manages the translation of `Element` objects to their `IString` form. It can provide strings for collection elements that are used in the `ICollectionViewListBox` part or `ICollectionViewComboBox` part.

To use the `IStringGenerator`, you must define a subclass of the `IStringGeneratorFn<Element>` template class and override the pure virtual function: `virtual IString stringFor(Element const& pElement)` . The

IStringGeneratorFn template class is an abstract base class that defines the protocol for storing and calling functions that generate IString objects. Objects of this class represent functions that are called when the stringFor function is called. The stringFor function accepts an object reference of the template class type.

It is a good idea to override the string generator of a collection list box when using the collection list box in tandem with Data Access Builder parts. In effect, you do not have to edit the source code generated by Data Access Builder to customize the contents of the collection list box.

To test the string generator class and display a list of property identifiers in an ICollectionViewListBox, use the **TinyApp** sample that you develop in Section “Using Data Access Builder Parts with Visual Builder” on page 140:

1. Build a new class, IStringGeneratorForPropertyFn, to set a new IStringGenerator for the ICollectionViewListBox (see Figure 72).
2. Open the settings notebook of the ICollectionViewListBox\* part.
3. Switch to the General page and fill in the String generator entry field with: **IStringGenerator<Property\*>(new IStringGeneratorForPropertyFn())**.
4. Switch to the Class Editor and, in the *Required include files* list box, add the name of the file where IStringGeneratorForPropertyFn is defined.

Now you can regenerate the code and compile it. When you run TinyApp, the ICollectionViewListBox should display the list of the property identifiers stored in the REAL database.



```
// Class used to set a new IStringGenerator for a DAX part when
// used with an ICollectionViewListBox

#include <PropertyV.hpp>    // DAX generated file.
                          // Property is the data object

#include <istrngen.hpp>     // header file for IStringGenerator class

class IStringGeneratorForPropertyFn :
    public IStringGeneratorFn<Property*>
{
    IStringGeneratorForPropertyFn() {};
    virtual ~IStringGeneratorForPropertyFn() {};

    virtual IString stringFor(Property* const& pProperty)
    {
        // Return the identifier of the property
        return pProperty->Property_id();
    }
}; // IStringGeneratorForPropertyFn
```

**Figure 72.** IStringGeneratorForPropertyFn Declaration


To build APropertySearchParameterView, follow the steps in Table 22.

Table 22. (Part 1 of 5) Building APropertySearchParameterView													
Step	Action												
1	Start Visual Builder from the Property project (select <b>Project</b> → <b>Visual</b> in the menu bar).												
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.												
3	<p>Fill in the entry fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Class name</td><td>APropertySearchParameterView</td></tr> <tr> <td>Description</td><td>View to collect the buyer preferences</td></tr> <tr> <td>File name</td><td>VRPROP</td></tr> <tr> <td>Part type</td><td>Visual part</td></tr> <tr> <td>Base class</td><td>IFrameWindow</td></tr> </table> <p>Click on the <b>Open</b> push button. An IFrameWindow* part is displayed on the free-form surface.</p>	Field	Value	Class name	APropertySearchParameterView	Description	View to collect the buyer preferences	File name	VRPROP	Part type	Visual part	Base class	IFrameWindow
Field	Value												
Class name	APropertySearchParameterView												
Description	View to collect the buyer preferences												
File name	VRPROP												
Part type	Visual part												
Base class	IFrameWindow												
4	Change the IFrameWindow* part title to <b>Search Property</b> .												
5	Delete the ICanvas* part in the IFrameWindow* part.												




**Table 22.** (Part 2 of 5) Building APropertySearchParameterView

Step	Action												
6	Add an IMultiCellCanvas* part in the IFrameWindow* part.												
7	<p>Open the settings notebook of the IMultiCellCanvas* part and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 12</li> <li><input type="checkbox"/> Number of columns: 7</li> <li><input type="checkbox"/> Expandable rows: Nos. 1, 3, 5, 7, 9, 11</li> <li><input type="checkbox"/> Expandable columns: Nos. 1, 5</li> </ul> <p>Close the settings notebook.</p>												
8	<p>Add five ICheckBox* parts, <b>A</b>, to cells (2, 2), (4, 2), (6, 2), (8, 2), and (10, 2) of the IMultiCellCanvas* part (ICheckBox* part is located in the <i>Buttons</i> category) and set their settings as follows:</p> <table> <tr> <th>Name</th><th>Text</th></tr> <tr> <td>CheckBoxArea</td><td>Area</td></tr> <tr> <td>CheckBoxPrice</td><td>Price Range</td></tr> <tr> <td>CheckBoxSize</td><td>Size Range</td></tr> <tr> <td>CheckBoxBedrooms</td><td>Bedrooms</td></tr> <tr> <td>CheckBoxBathrooms</td><td>Bathrooms</td></tr> </table>	Name	Text	CheckBoxArea	Area	CheckBoxPrice	Price Range	CheckBoxSize	Size Range	CheckBoxBedrooms	Bedrooms	CheckBoxBathrooms	Bathrooms
Name	Text												
CheckBoxArea	Area												
CheckBoxPrice	Price Range												
CheckBoxSize	Size Range												
CheckBoxBedrooms	Bedrooms												
CheckBoxBathrooms	Bathrooms												
9	Add an ICollectionViewListBox* part, <b>B</b> , to the IMultiCellCanvas* part and make it span three columns from cell (2, 4) to cell (2, 6).												
10	<p>Open the settings notebook of the ICollectionViewListBox* part and set the fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Name</td><td>CollectionViewArea*</td></tr> <tr> <td>Item type</td><td>List_area*</td></tr> <tr> <td>Limit</td><td>20</td></tr> <tr> <td>Combo box type</td><td>Read-only drop-down</td></tr> </table> <p>The area is 20 characters wide (see the PROPERTY table structure in Appendix C, “Database Definition,” on page 515). The list box is set to read-only to prevent users from typing in a property area that does not exist in the database.</p>	Field	Value	Name	CollectionViewArea*	Item type	List_area*	Limit	20	Combo box type	Read-only drop-down		
Field	Value												
Name	CollectionViewArea*												
Item type	List_area*												
Limit	20												
Combo box type	Read-only drop-down												
11	Switch to the Control page of the ICollectionViewListBox* part settings notebook and deselect the <b>Enabled</b> check box. In “APropertySearchParameterView” on page 307, you will draw connections so that this control is enabled only when the CheckBoxArea is selected. Close the settings notebook.												
12	Add four IEntryField* parts to cells (4, 4), (4, 6), (6, 4), and (6, 6) of the IMultiCellCanvas* part to hold the price and size ranges, <b>C</b> , <b>D</b> , <b>E</b> , <b>F</b> .												

**Table 22.** (Part 3 of 5) Building APropertySearchParameterView

Step	Action																
13	<p>Open the settings notebooks of each entry field and configure them as follows:</p> <table><thead><tr><th>Name</th><th>Limit</th><th>Enabled check box</th></tr></thead><tbody><tr><td>EntryFieldMinPrice</td><td>7</td><td>unselect</td></tr><tr><td>EntryFieldMaxPrice</td><td>7</td><td>unselect</td></tr><tr><td>EntryFieldMinSize</td><td>5</td><td>unselect</td></tr><tr><td>EntryFieldMaxSize</td><td>5</td><td>unselect</td></tr></tbody></table> <p>Close the settings notebooks. Deselect the <b>Enabled</b> check box of each entry field from the Control page of its settings notebook. In “APropertySearchParameterView” on page 307, you will draw connections so that this control is enabled only when their corresponding check box (CheckBoxPrice or CheckBoxSize) is selected.</p>	Name	Limit	Enabled check box	EntryFieldMinPrice	7	unselect	EntryFieldMaxPrice	7	unselect	EntryFieldMinSize	5	unselect	EntryFieldMaxSize	5	unselect	
Name	Limit	Enabled check box															
EntryFieldMinPrice	7	unselect															
EntryFieldMaxPrice	7	unselect															
EntryFieldMinSize	5	unselect															
EntryFieldMaxSize	5	unselect															
14	<p>Add an ISetCanvas* part in cell (8, 4) of the IMultiCellCanvas* part and set its margin width and height to 0. This setting enables the numeric spin button boundaries for the number of bedrooms to line up with the set canvas boundaries.</p>																
15	<p>Add an INumericSpinButton* part to the ISetCanvas* part to hold the number of bedrooms, , and configure it as follows:</p> <table><thead><tr><th>Field</th><th>Value</th></tr></thead><tbody><tr><td>Name</td><td>NumericSpinButtonBedrooms</td></tr><tr><td>Alignment</td><td>Center</td></tr><tr><td>Limit</td><td>1</td></tr><tr><td>Lower</td><td>0</td></tr><tr><td>Upper</td><td>6</td></tr><tr><td>Value</td><td>0</td></tr><tr><td>Enabled</td><td>unselect</td></tr></tbody></table> <p>Deselect the <b>Enabled</b> check box of the numeric spin button from its settings notebook Control page. In “APropertySearchParameterView” on page 307, you will draw connections so that this control is enabled only when the corresponding CheckBoxBedrooms check box is selected.</p>	Field	Value	Name	NumericSpinButtonBedrooms	Alignment	Center	Limit	1	Lower	0	Upper	6	Value	0	Enabled	unselect
Field	Value																
Name	NumericSpinButtonBedrooms																
Alignment	Center																
Limit	1																
Lower	0																
Upper	6																
Value	0																
Enabled	unselect																
16	<p>Add an ISetCanvas* part in cell (10, 4) of the IMultiCellCanvas* part and set its margin width and height to 0. This setting enable the numeric spin button for the number of bathrooms to line up with the set canvas boundaries.</p>																

**Table 22.** (Part 4 of 5) Building APropertySearchParameterView

Step	Action																
17	<p>Add an INumericSpinButton* part to the ISetCanvas* part to hold the number of bathrooms, , and configure them as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Name</td><td>NumericSpinButtonBathrooms</td></tr> <tr> <td>Alignment</td><td>Center</td></tr> <tr> <td>Limit</td><td>1</td></tr> <tr> <td>Lower</td><td>1</td></tr> <tr> <td>Upper</td><td>4</td></tr> <tr> <td>Value</td><td>1</td></tr> <tr> <td>Enabled</td><td>unselect</td></tr> </table> <p>Deselect the <b>Enabled</b> check box of the numeric spin button from its Settings notebook Control page. In “APropertySearchParameterView” on page 307, you will draw connections so that this control is enabled only when the corresponding CheckBoxBathrooms check box is selected.</p>	Field	Value	Name	NumericSpinButtonBathrooms	Alignment	Center	Limit	1	Lower	1	Upper	4	Value	1	Enabled	unselect
Field	Value																
Name	NumericSpinButtonBathrooms																
Alignment	Center																
Limit	1																
Lower	1																
Upper	4																
Value	1																
Enabled	unselect																
18	<p>Add an ISetCanvas* part to the IMultiCellCanvas* part and make it span from cell (12, 2) to cell (12, 5) as shown in Figure 71 on page 201, .</p>																
19	<p>Add three IPushButton* parts to the ISetCanvas* part and set their names as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> <b>PushButtonSearch</b></li> <li><input type="checkbox"/> <b>PushButtonCancel</b></li> <li><input type="checkbox"/> <b>PushButtonHelp</b></li> </ul>																
20	<p>Change the <i>text</i> attribute of the three push buttons as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> <b>~Search</b> for PushButtonSearch</li> <li><input type="checkbox"/> <b>~Cancel</b> for PushButtonCancel</li> <li><input type="checkbox"/> <b>~Help</b> for PushButtonHelp</li> </ul> <p>Notice the use of ~ for the key accelerator.</p>																
21	<p>Open the settings notebook of PushButtonHelp and switch to the Styles page. Set the <i>help</i> radio button to <b>On</b> to turn this regular push button into a help push button. Set the <i>noPointerFocus</i> radio button to <b>On</b> to prevent the help push button from getting the input focus when the user clicks on it. Close the settings notebook.</p>																
22	<p>Add an IInfoArea* part to the IFrameWindow*, .</p>																

**Table 22.** (Part 5 of 5) Building APropertySearchParameterView

Step	Action																																																
23	<p>Set the Tabbing and Depth Order of the dialog box and define six tabbing groups as follows (see “Setting the Tabbing and Depth Order” on page 157):</p> <table><tr><th>Group</th><th>Tab</th><th>Feature</th></tr><tr><td>X</td><td>X</td><td>CheckBoxArea</td></tr><tr><td>-</td><td>X</td><td>CollectionViewArea</td></tr><tr><td>X</td><td>X</td><td>CheckBoxSize</td></tr><tr><td>-</td><td>X</td><td>EntryFieldMinSize</td></tr><tr><td>-</td><td>X</td><td>EntryFieldMaxSize</td></tr><tr><td>X</td><td>X</td><td>CheckBoxPrice</td></tr><tr><td>-</td><td>X</td><td>EntryFieldMinPrice</td></tr><tr><td>-</td><td>X</td><td>EntryFieldMaxPrice</td></tr><tr><td>X</td><td>X</td><td>CheckBoxBedrooms</td></tr><tr><td>-</td><td>X</td><td>NumericSpinButtonBedrooms</td></tr><tr><td>X</td><td>X</td><td>CheckBoxBathrooms</td></tr><tr><td>-</td><td>X</td><td>NumericSpinButtonBathrooms</td></tr><tr><td>X</td><td>X</td><td>PushButtonSearch</td></tr><tr><td>-</td><td>X</td><td>PushButtonCancel</td></tr><tr><td>-</td><td>X</td><td>PushButtonHelp</td></tr></table>	Group	Tab	Feature	X	X	CheckBoxArea	-	X	CollectionViewArea	X	X	CheckBoxSize	-	X	EntryFieldMinSize	-	X	EntryFieldMaxSize	X	X	CheckBoxPrice	-	X	EntryFieldMinPrice	-	X	EntryFieldMaxPrice	X	X	CheckBoxBedrooms	-	X	NumericSpinButtonBedrooms	X	X	CheckBoxBathrooms	-	X	NumericSpinButtonBathrooms	X	X	PushButtonSearch	-	X	PushButtonCancel	-	X	PushButtonHelp
Group	Tab	Feature																																															
X	X	CheckBoxArea																																															
-	X	CollectionViewArea																																															
X	X	CheckBoxSize																																															
-	X	EntryFieldMinSize																																															
-	X	EntryFieldMaxSize																																															
X	X	CheckBoxPrice																																															
-	X	EntryFieldMinPrice																																															
-	X	EntryFieldMaxPrice																																															
X	X	CheckBoxBedrooms																																															
-	X	NumericSpinButtonBedrooms																																															
X	X	CheckBoxBathrooms																																															
-	X	NumericSpinButtonBathrooms																																															
X	X	PushButtonSearch																																															
-	X	PushButtonCancel																																															
-	X	PushButtonHelp																																															
<b>Note:</b> Reverse highlighted letters are keyed to Figure 71 on page 201.																																																	

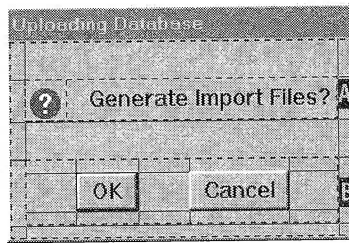
Now you can save APropertySearchParameterView. First, switch to the Class Editor and fill in the *Code generation file* group box as follows:

- ☐ C++ header file (.hpp): **vrpsrcv.hpp**
- ☐ C++ code file (.cpp): **vrpsrcv.cpp**

Then, save the part.

## AUploadView

AUploadView (Figure 73) is a dialog window that enables the user to generate DB2 for Windows import files. These import files can be sent to the real estate agency's server and then uploaded to update its database.



**Figure 73.** AUploadView

To build AUploadView, follow the steps in Table 23.

<b>Table 23.</b> (Part 1 of 3) Building AUploadView													
<b>Step</b>	<b>Action</b>												
1	Start Visual Builder from the Services project (select <b>Project</b> → <b>Visual</b> in the menu bar).												
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.												
3	<p>Fill in the entry fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Class name</td><td>AUploadView</td></tr> <tr> <td>Description</td><td>General view to generate DB2 import files</td></tr> <tr> <td>File name</td><td>VRSERV</td></tr> <tr> <td>Part type</td><td>visual part</td></tr> <tr> <td>Base class</td><td>IFrameWindow</td></tr> </table> <p>Click on the <b>Open</b> push button. An IFrameWindow* part is displayed on the free-form surface.</p>	Field	Value	Class name	AUploadView	Description	General view to generate DB2 import files	File name	VRSERV	Part type	visual part	Base class	IFrameWindow
Field	Value												
Class name	AUploadView												
Description	General view to generate DB2 import files												
File name	VRSERV												
Part type	visual part												
Base class	IFrameWindow												
4	<p>Open the settings notebook of the frame window and change the style attributes as follows:</p> <table> <tr> <th>Attribute</th><th>Setting</th></tr> <tr> <td>dialogBorder</td><td>On</td></tr> <tr> <td>maximizeButton</td><td>Off</td></tr> <tr> <td>minimizeButton</td><td>Off</td></tr> <tr> <td>sizingBorder</td><td>Off</td></tr> <tr> <td>systemMenu</td><td>Off</td></tr> </table> <p>Close the settings notebook. The window becomes a nonresizable dialog box.</p>	Attribute	Setting	dialogBorder	On	maximizeButton	Off	minimizeButton	Off	sizingBorder	Off	systemMenu	Off
Attribute	Setting												
dialogBorder	On												
maximizeButton	Off												
minimizeButton	Off												
sizingBorder	Off												
systemMenu	Off												
5	Change the IFrameWindow* part title to <b>Uploading Database</b> .												
6	Delete the ICanvas* part from the IFrameWindow* part.												
7	Add an IMultiCellCanvas* part to the IFrameWindow* part.												

**Table 23.** (Part 2 of 3) Building AUploadView

Step	Action									
8	<p>Open the settings notebook of the IMultiCellCanvas* part and configure it as follows:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> Number of rows: 5</li><li><input type="checkbox"/> Number of columns: 3</li><li><input type="checkbox"/> Expandable rows: Nos. 1, 3</li><li><input type="checkbox"/> Expandable columns: Nos. 1, 3</li></ul> <p>Close the settings notebook.</p>									
9	<p>Switch to the Color page and select the <b>Colors</b> radio button of the <i>Color selection</i> group box. Then select <b>paleGray</b> in the <i>Color values</i> drop-down list box. Close the settings notebook.</p>									
10	<p>Add an ISetCanvas* part, <b>A</b>, to cell (2, 2) of the IMultiCellCanvas* part. This set canvas will hold an icon and a static text control.</p>									
11	<p>Add an IIconControl* part to the ISetCanvas* part.</p>									
12	<p>Open the settings notebook of the IIconControl* part. Set the DLL name to <b>realicon</b> and the resource ID to <b>502</b>.</p>									
13	<p>Add an IStaticText* part to the ISetCanvas* part and change its labels to <b>Generate Import Files</b>.</p>									
14	<p>Open the settings notebook of the ISetCanvas* part, set its alignment to center (middle radio button), and adjust the width and height of the <i>Margin</i> and <i>Pad</i> group boxes as follows:</p> <table><thead><tr><th>Group box</th><th>Width</th><th>Height</th></tr></thead><tbody><tr><td>Margin</td><td>0</td><td>0</td></tr><tr><td>Pad</td><td>10</td><td>0</td></tr></tbody></table> <p>Close the settings notebook.</p>	Group box	Width	Height	Margin	0	0	Pad	10	0
Group box	Width	Height								
Margin	0	0								
Pad	10	0								
15	<p>Add an IMultiCellCanvas* part, <b>B</b>, to cell (4, 2) of the IMultiCellCanvas* part and configure it as follows:</p> <ul style="list-style-type: none"><li><input type="checkbox"/> Number of rows: 3</li><li><input type="checkbox"/> Number of columns: 5</li><li><input type="checkbox"/> Expandable rows: none</li><li><input type="checkbox"/> Expandable columns: Nos. 1, 3, 5</li></ul>									
16	<p>Add two IPushButton* parts to cells (2, 2) and (2, 4) of the IMultiCellCanvas* part <b>B</b> and change their labels as shown in Figure 65 on page 183.</p>									
17	<p>Change the push button names to <b>PushButtonOK</b> and <b>PushButtonCancel</b>.</p>									
18	<p>Promote the <b>buttonClickEvent</b> event of <i>PushbuttonOK</i>.</p>									

**Table 23.** (Part 3 of 3) Building AUpLoadView

Step	Action									
19	Set <b>PushButtonOK</b> as the default push button (set defaultButton to <b>On</b> in the Styles page of the settings notebook).									
20	<p>Set the Tabbing and Depth Order of the two push buttons as follows (see “Setting the Tabbing and Depth Order” on page 157):</p> <table><thead><tr><th>Group</th><th>Tab</th><th>Feature</th></tr></thead><tbody><tr><td>-</td><td>X</td><td>PushButtonOK</td></tr><tr><td>-</td><td>X</td><td>PushButtonCancel</td></tr></tbody></table> <p>Close the dialog box.</p>	Group	Tab	Feature	-	X	PushButtonOK	-	X	PushButtonCancel
Group	Tab	Feature								
-	X	PushButtonOK								
-	X	PushButtonCancel								
<b>Note:</b> Reverse highlighted letters are keyed to Figure 73 on page 209.										

Now you can save AUpLoadView. First, switch to the Class Editor and fill in the *Code generation file* group box as follows:

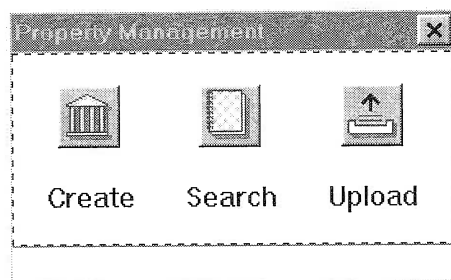
- ☐ C++ header file (.hpp): **vrsuplv.hpp**
- ☐ C++ code file (.cpp): **vrsuplv.cpp**

Then, save the part.

## APropertyManagementView

APropertyManagementView (Figure 74) is the first window that the user sees when accessing the Property subsystem. It calls the other views you have built previously. By clicking on graphical push buttons, the user can select the following options:

- ☐ Create a property (APropertyCreateView)
- ☐ Search properties (APropertySearchParameterView)
- ☐ Upload property tables (AUpLoadView)

**Figure 74.** APropertyManagementView

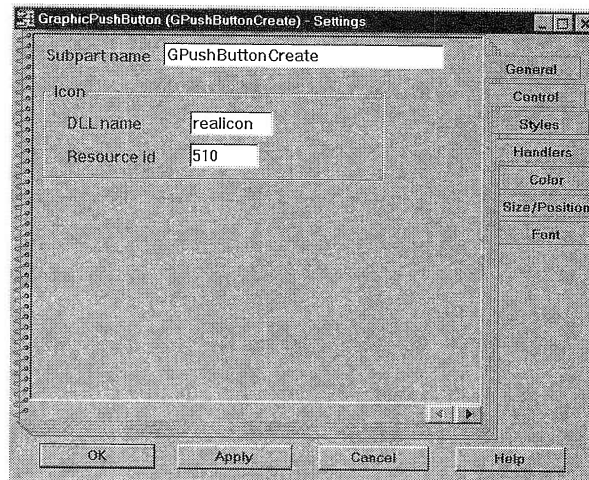


APropertyManagementView derives from an IFrameWindow part. It is set to be nonresizable. Because each control is lined up along two rows and three columns, the set canvas appears to be the canvas to use to hold the different controls. In addition, the set canvas is a minimum-size canvas and supports multiple screen resolutions and font settings. To ensure that the set canvas stays centered in the middle of the client area, use a small multicell canvas of three rows and three columns for the client area of the frame window. Set the multicell canvas top row, bottom row, left column, and right column to be expandable and place the set canvas in the middle of the multicell canvas: cell (2, 2).

## Using Graphic Push Buttons

An IGraphicPushButton part is a selection button for an action or routing choice. The choice is represented by a graphical image on the push button. The user can click on the push button to perform an action. Usually, you can use an IGraphicPushButton part whenever an action can be represented graphically.

To associate a graphical image with the push button, enter the *DLL name* in the *DLL name* field and the resource ID in the *Resource id* field of the IGraphicPushButton General settings page (Figure 75).



**Figure 75.** IGraphicPushButton General Settings Page

To build APropertyManagementView, follow the steps in Table 24.

<b>Table 24.</b> (Part 1 of 2) Building APropertyManagementView													
<b>Step</b>	<b>Action</b>												
1	Start Visual Builder from the Property project (select <b>Project</b> → <b>Visual</b> in the menu bar).												
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.												
3	<p>Fill in the entry fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Class name</td><td>APropertyManagementView</td></tr> <tr> <td>Description</td><td>Main Window for the property subsystem</td></tr> <tr> <td>File name</td><td>VRPROP</td></tr> <tr> <td>Part type</td><td>visual part</td></tr> <tr> <td>Base class</td><td>IFrameWindow</td></tr> </table> <p>Click on the <b>Open</b> push button. An IFrameWindow* part is displayed on the free-form surface.</p>	Field	Value	Class name	APropertyManagementView	Description	Main Window for the property subsystem	File name	VRPROP	Part type	visual part	Base class	IFrameWindow
Field	Value												
Class name	APropertyManagementView												
Description	Main Window for the property subsystem												
File name	VRPROP												
Part type	visual part												
Base class	IFrameWindow												
4	<p>Open the settings notebook of the frame window and change the style attributes as follows:</p> <table> <tr> <th>Attribute</th><th>Setting</th></tr> <tr> <td>dialogBorder</td><td>On</td></tr> <tr> <td>maximizeButton</td><td>Off</td></tr> <tr> <td>minimizeButton</td><td>Off</td></tr> <tr> <td>sizingBorder</td><td>Off</td></tr> </table> <p>Close the settings notebook. The window becomes a nonresizable dialog box. Notice that we keep the system menu to enable the user to close the window.</p>	Attribute	Setting	dialogBorder	On	maximizeButton	Off	minimizeButton	Off	sizingBorder	Off		
Attribute	Setting												
dialogBorder	On												
maximizeButton	Off												
minimizeButton	Off												
sizingBorder	Off												
5	Delete the ICanvas* part in the IFrameWindow* part.												
6	<p>Add an IMultiCellCanvas* part to the IFrameWindow* part and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 3</li> <li><input type="checkbox"/> Number of columns: 3</li> <li><input type="checkbox"/> Expandable rows: Nos. 1, 3</li> <li><input type="checkbox"/> Expandable columns: Nos. 1, 3</li> </ul>												
7	Change the IFrameWindow* part title to <b>Property Management</b> .												

**Table 24.** (Part 2 of 2) Building APropertyManagementView

Step	Action																		
8	<p>Add an ISetCanvas* part to cell (2, 2) of the IMultiCellCanvas* part and configure it as follows:</p> <table><tr><th>Field</th><th>Value</th></tr><tr><td>Deck Orientation</td><td>Horizontal</td></tr><tr><td>Pack Type</td><td>Even</td></tr><tr><td>Deck Count</td><td>2</td></tr><tr><td>Margin Width</td><td>20</td></tr><tr><td>Margin Height</td><td>20</td></tr><tr><td>Pad Width</td><td>30</td></tr><tr><td>Pad Height</td><td>20</td></tr><tr><td>Alignment</td><td>Centered</td></tr></table> <p>Use the <b>Apply</b> push button of the settings notebook to adjust margins and pads before closing the notebook to effect the changes.</p>	Field	Value	Deck Orientation	Horizontal	Pack Type	Even	Deck Count	2	Margin Width	20	Margin Height	20	Pad Width	30	Pad Height	20	Alignment	Centered
Field	Value																		
Deck Orientation	Horizontal																		
Pack Type	Even																		
Deck Count	2																		
Margin Width	20																		
Margin Height	20																		
Pad Width	30																		
Pad Height	20																		
Alignment	Centered																		
9	<p>Add three IGraphicPushButton* parts to the client area and change their settings as follows (the IGraphicPushButton* part is located in the <i>Buttons</i> category):</p> <table><tr><th>Button</th><th>SubPart</th><th>DLL</th><th>Resource Id</th></tr><tr><td>Create</td><td>GPushButtonCreate</td><td>realicon</td><td>510</td></tr><tr><td>Search</td><td>GPushButtonSearch</td><td>realicon</td><td>505</td></tr><tr><td>Upload</td><td>GPushButtonUpload</td><td>realicon</td><td>503</td></tr></table>	Button	SubPart	DLL	Resource Id	Create	GPushButtonCreate	realicon	510	Search	GPushButtonSearch	realicon	505	Upload	GPushButtonUpload	realicon	503		
Button	SubPart	DLL	Resource Id																
Create	GPushButtonCreate	realicon	510																
Search	GPushButtonSearch	realicon	505																
Upload	GPushButtonUpload	realicon	503																
10	<p>Add three IStaticText* parts and change their <i>text</i> attributes to <b>Create</b>, <b>Search</b>, and <b>Upload</b>.</p>																		
<p><b>Note:</b> Because the number of decks in the set canvas defines an implicit order, you must add the three graphic push buttons first, then the three static text controls.</p>																			
11	<p>Add an IInfoArea* part to the IFrameWindow* part.</p>																		
12	<p>Set the Tabbing and Depth Order of the three graphic push buttons as follows (see “Setting the Tabbing and Depth Order” on page 157):</p> <table><tr><th>Group</th><th>Tab</th><th>Feature</th></tr><tr><td>-</td><td>X</td><td>GPushButtonCreate</td></tr><tr><td>-</td><td>X</td><td>GPushButtonSearch</td></tr><tr><td>-</td><td>X</td><td>GPushButtonUpload</td></tr></table> <p>Close the dialog box.</p>	Group	Tab	Feature	-	X	GPushButtonCreate	-	X	GPushButtonSearch	-	X	GPushButtonUpload						
Group	Tab	Feature																	
-	X	GPushButtonCreate																	
-	X	GPushButtonSearch																	
-	X	GPushButtonUpload																	

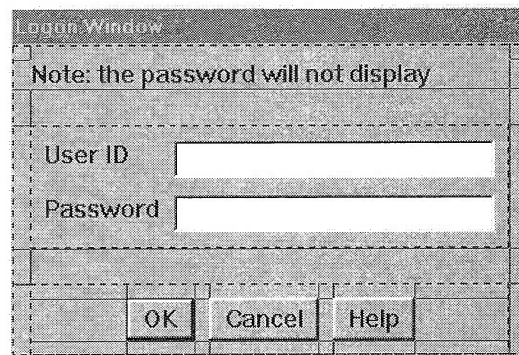
Now you can save APropertyManagementView. First, switch to the Class Editor and fill in the *Code generation file* group box as follows:

- ☐ C++ header file (.hpp): **vrpmngv.hpp**
- ☐ C++ code file (.cpp): **vrpmngv.cpp**

Then, save the part.

## ALogonView

ALogonView (Figure 76) is a general logon view that can be used to authenticate a user. When users start the application, they must first connect to the database to access the different subsystems. ALogonView prompts users for their user ID and password. This information is passed to an IDatastore part to establish the connection (see Chapter 6, “Mapping Relational Tables Using Data Access Builder,” on page 131).



**Figure 76.** ALogonView

ALogonView is simple; the only peculiarity is an event handler, which you attach to each entry field to force the user input to be in upper-case. We explain how to construct your own event handler in “Event Handler” on page 237.

To build ALogonView, follow the steps in Table 25.

<b>Table 25.</b> (Part 1 of 3) Building ALogonView	
<b>Step</b>	<b>Action</b>
1	Start Visual Builder from the Common project (select <b>Project</b> → <b>Visual</b> in the menu bar).
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.

**Table 25.** (Part 2 of 3) Building ALogonView

Step	Action												
3	<p>Fill in the entry fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Class name</td><td>ALogonView</td></tr> <tr> <td>Description</td><td>General purpose view to log on</td></tr> <tr> <td>File name</td><td>VRCOMM</td></tr> <tr> <td>Part type</td><td>visual part</td></tr> <tr> <td>Base class</td><td>IFrameWindow</td></tr> </table> <p>Click on the <b>Open</b> push button. An IFrameWindow* part is displayed on the free-form surface.</p>	Field	Value	Class name	ALogonView	Description	General purpose view to log on	File name	VRCOMM	Part type	visual part	Base class	IFrameWindow
Field	Value												
Class name	ALogonView												
Description	General purpose view to log on												
File name	VRCOMM												
Part type	visual part												
Base class	IFrameWindow												
4	<p>Open the settings notebook of the frame window and change the style attributes as follows:</p> <table> <tr> <th>Attribute</th><th>Setting</th></tr> <tr> <td>dialogBorder</td><td>On</td></tr> <tr> <td>maximizeButton</td><td>Off</td></tr> <tr> <td>minimizeButton</td><td>Off</td></tr> <tr> <td>sizingBorder</td><td>Off</td></tr> <tr> <td>systemMenu</td><td>Off</td></tr> </table> <p>Close the settings notebook. The window becomes a nonresizable dialog box.</p>	Attribute	Setting	dialogBorder	On	maximizeButton	Off	minimizeButton	Off	sizingBorder	Off	systemMenu	Off
Attribute	Setting												
dialogBorder	On												
maximizeButton	Off												
minimizeButton	Off												
sizingBorder	Off												
systemMenu	Off												
5	Change the IFrameWindow* part title to <b>Logon Window</b> .												
6	Delete the ICanvas* part in the IFrameWindow* part.												
7	<p>Add an IMultiCellCanvas* part to the IFrameWindow* part and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 6</li> <li><input type="checkbox"/> Number of columns: 3</li> <li><input type="checkbox"/> Expandable rows: Nos. 3, 5</li> <li><input type="checkbox"/> Expandable columns: Nos. 1, 3</li> </ul>												
8	Add an IStaticText* part to cell (2, 2) of the IMultiCellCanvas* and change its label to <b>Note: the password will not display</b> .												
9	<p>Add an ISetCanvas* part in cell (4, 2) of the IMultiCellCanvas* part and change the setting of its attributes as follows:</p> <table> <tr> <th>Attribute</th><th>Setting</th></tr> <tr> <td>Deck Orientation</td><td>Vertical</td></tr> <tr> <td>Pack Type</td><td>Even</td></tr> <tr> <td>Deck Count</td><td>2</td></tr> </table>	Attribute	Setting	Deck Orientation	Vertical	Pack Type	Even	Deck Count	2				
Attribute	Setting												
Deck Orientation	Vertical												
Pack Type	Even												
Deck Count	2												
10	Add two IStaticText* parts to the ISetCanvas* part and change their labels to <b>User ID</b> and <b>Password</b> .												

**Table 25.** (Part 3 of 3) Building ALogonView

Step	Action																		
11	Add two IEntryField* parts and change their names to <b>EntryFieldUserID</b> and <b>EntryFieldPassword</b> and their limit to <b>20</b> . Reset their size to the default size, using the <b>Reset to default size</b> action from their pop-up menu.																		
12	Promote each <i>text</i> attribute of each IEntryField* part. Because ALogonView will be reused in other parts, you must promote these features to have access to the logon information.																		
13	Add an IMultiCellCanvas* part to cell (6, 2) of the first IMultiCellCanvas* part and configure it as follows: <div><input type="checkbox"/> Number of rows: 3</div> <div><input type="checkbox"/> Number of columns: 7</div> <div><input type="checkbox"/> Expandable rows: none</div> <div><input type="checkbox"/> Expandable columns: Nos. 1, 7</div>																		
14	Add three IPushButton* parts and the new IMultiCellCanvas* part in cells (2, 2), (2, 4), and (2, 6) and change their <i>text</i> labels, respectively, to <b>OK</b> , <b>Cancel</b> , and <b>Help</b> (Figure 76 on page 215).																		
15	Change the name of the IPushButton* parts to <b>PushButtonOK</b> , <b>PushButtonCancel</b> , and <b>PushButtonHelp</b> .																		
16	Open the settings notebook of EntryFieldPassword and, on the Handlers page, add UpperCaseKbdHandler. Then switch to the Styles page and set the <i>unreadable</i> style to <b>On</b> . Close the settings notebook.																		
17	Open the settings notebook of PushButtonHelp and switch to the Styles page. Set the <i>help</i> radio button to <b>On</b> . Set the <i>noPointerFocus</i> radio button to <b>On</b> . Close the settings notebook.																		
18	Set <b>PushButtonOK</b> as the default push button (set defaultButton to <b>On</b> in the Styles page of the settings notebook).																		
19	<div>Set the Tabbing and Depth Order dialog box of the controls and define two tabbing groups as follows:</div> <table><thead><tr><th>Group</th><th>Tab</th><th>Feature</th></tr></thead><tbody><tr><td>X</td><td>X</td><td>EntryFieldUserID</td></tr><tr><td>-</td><td>X</td><td>EntryFieldPassword</td></tr><tr><td>X</td><td>X</td><td>PushButtonOK</td></tr><tr><td>-</td><td>X</td><td>PushButtonCancel</td></tr><tr><td>-</td><td>X</td><td>PushButtonHelp</td></tr></tbody></table> <div>Close the dialog box.</div>	Group	Tab	Feature	X	X	EntryFieldUserID	-	X	EntryFieldPassword	X	X	PushButtonOK	-	X	PushButtonCancel	-	X	PushButtonHelp
Group	Tab	Feature																	
X	X	EntryFieldUserID																	
-	X	EntryFieldPassword																	
X	X	PushButtonOK																	
-	X	PushButtonCancel																	
-	X	PushButtonHelp																	

Now you can save ALogonView. First, switch to the Class Editor and fill in the *Code generation file* group box as follows:

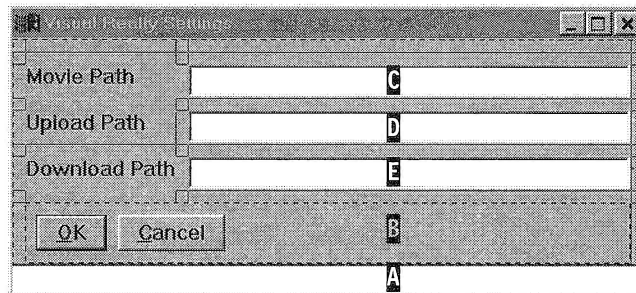
- ☐ C++ header file (.hpp): **vrclogv.hpp**
- ☐ C++ code file (.cpp): **vrclogv.cpp**

Then, save the part.

## ARealSettingsView

ARealSettingsView (Figure 77) enables the user to update the Visual Realty application's settings. The settings consist of the directory information where files are uploaded or downloaded from the server:

- Movie path**      The path where the video files are placed when they are downloaded from the server
- Upload path**      The path where the export files are produced to be uploaded to the server
- Download path**    The path where the import files are placed when they are downloaded from the server



**Figure 77.** ARealSettingsView

An IProfile part, which stores the directory information, works in tandem with ARealSettingsView. The use of IProfile part is described in "ARealSettingsView" on page 331.

ARealSettingsView is not reused in other parts of the application, so you do not have to promote its features.

To build ARealSettingsView, follow the steps in Table 26.

<b>Table 26.</b> (Part 1 of 2) Building ARealSettingsView													
<b>Step</b>	<b>Action</b>												
1	Start Visual Builder from the Common project (select <b>Project</b> → <b>Visual</b> in the menu bar).												
2	From the Visual Builder window, select <b>Part</b> → <b>New...</b> option.												
3	<p>Fill in the entry fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Class name</td><td>ARealSettingsView</td></tr> <tr> <td>Description</td><td>Application settings view</td></tr> <tr> <td>File name</td><td>VRCOMM</td></tr> <tr> <td>Part type</td><td>visual part</td></tr> <tr> <td>Base class</td><td>IFrameWindow</td></tr> </table> <p>Click on the <b>Open</b> push button. An IFrameWindow* part is displayed on the free-form surface.</p>	Field	Value	Class name	ARealSettingsView	Description	Application settings view	File name	VRCOMM	Part type	visual part	Base class	IFrameWindow
Field	Value												
Class name	ARealSettingsView												
Description	Application settings view												
File name	VRCOMM												
Part type	visual part												
Base class	IFrameWindow												
4	Change the IFrameWindow* part title to <b>Visual Realty Settings</b> .												
5	Delete the ICanvas* part from the IFrameWindow* part.												
6	Add an IMultiCellCanvas* part in the IFrameWindow* part.												
7	<p>Open the settings notebook of the IMultiCellCanvas* part and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 10</li> <li><input type="checkbox"/> Number of columns: 5</li> <li><input type="checkbox"/> Expandable row: No. 8</li> <li><input type="checkbox"/> Expandable column: No. 4</li> </ul> <p>Close the settings notebook.</p>												
8	Add an IInfoArea* part to the IFrameWindow* part, <b>A</b> .												
9	Add an ISetCanvas* part to the IMultiCellCanvas* part as shown on Figure 77 on page 218, <b>B</b> .												
10	<p>Add two IPushButton* parts to the ISetCanvas* part and set their names as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> <b>PushButtonOK</b></li> <li><input type="checkbox"/> <b>PushButtonCancel</b></li> </ul>												
11	<p>Change their labels as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> <b>~OK</b> for PushButtonOK</li> <li><input type="checkbox"/> <b>~Cancel</b> for PushButtonCancel</li> </ul> <p>Notice the use of ~ for the key accelerator.</p>												



Table 26. (Part 2 of 2) Building ARealSettingsView																				
Step	Action																			
12	Add three IEntryField* parts and change their names to <b>EntryFieldMovie</b> , <b>EntryFieldUpload</b> , and <b>EntryFieldDownload</b> , <b>C</b> , <b>D</b> , <b>E</b> . Set their limit to <b>50</b> characters.																			
13	Set <b>PushbuttonOK</b> as the default push button (set defaultButton to <b>On</b> in the Styles page of the settings notebook).																			
14	Set the Tabbing and Depth Order and define two tabbing groups for the controls as follows: <table><tr><th>Group</th><th>Tab</th><th>Feature</th></tr><tr><td>X</td><td>X</td><td>EntryFieldMovie</td></tr><tr><td>-</td><td>X</td><td>EntryFieldUpload</td></tr><tr><td>-</td><td>X</td><td>EntryFieldDownload</td></tr><tr><td>X</td><td>X</td><td>PushButtonOK</td></tr><tr><td>-</td><td>X</td><td>PushButtonCancel</td></tr></table> Close the dialog box.		Group	Tab	Feature	X	X	EntryFieldMovie	-	X	EntryFieldUpload	-	X	EntryFieldDownload	X	X	PushButtonOK	-	X	PushButtonCancel
Group	Tab	Feature																		
X	X	EntryFieldMovie																		
-	X	EntryFieldUpload																		
-	X	EntryFieldDownload																		
X	X	PushButtonOK																		
-	X	PushButtonCancel																		
<b>Note:</b> Reverse highlighted letters are keyed to Figure 77 on page 218.																				

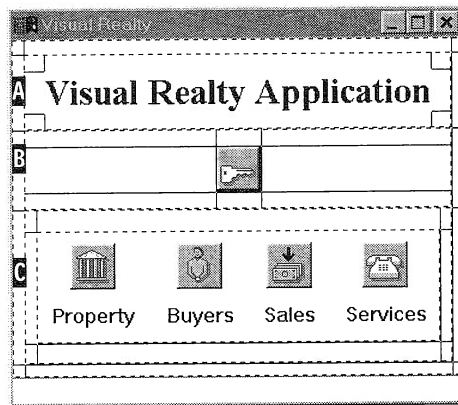
Now you can save ARealSettingsView. First, switch to the Class Interface Editor and fill in the *Code generation file* group box as follows:

- ☐ C++ header file (.hpp): **vrcsetv.hpp**
- ☐ C++ code file (.cpp): **vrcsetv.cpp**

Then, save the part.

## ARealMainView

ARealMainView (Figure 78) is the application main view. From this view, the user can log on to the database and access each subsystem.



**Figure 78.** ARealMainView

ARealMainView derives from an IFrameWindow part. It is built from the parts that you already know: IMultiCellCanvas, ISetCanvas, IGraphicPushButton, and IStaticText. The user can resize the window; the IMultiCellCanvas is used intensively to enable the controls to distribute evenly.

To build ARealMainView, follow the steps in Table 27.

<b>Table 27.</b> (Part 1 of 3) Building ARealMainView													
<b>Step</b>	<b>Action</b>												
1	Start Visual Builder from the Visual Realty project (select <b>Project</b> → <b>Visual</b> in the menu bar).												
2	Create a visual part by using the following settings: <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>Class name</td><td>ARealMainView</td></tr> <tr> <td>Description</td><td>Application main view</td></tr> <tr> <td>File name</td><td>VRMAIN</td></tr> <tr> <td>Part type</td><td>visual part</td></tr> <tr> <td>Base class</td><td>IFrameWindow</td></tr> </table>	Field	Value	Class name	ARealMainView	Description	Application main view	File name	VRMAIN	Part type	visual part	Base class	IFrameWindow
Field	Value												
Class name	ARealMainView												
Description	Application main view												
File name	VRMAIN												
Part type	visual part												
Base class	IFrameWindow												
3	Change the title of the IFrameWindow* part to <b>Visual Realty</b> .												
4	Add an IInfoArea* part to the IFrameWindow* part.												
5	Delete the ICanvas* part from the IFrameWindow* part.												

**Table 27.** (Part 2 of 3) Building ARealMainView

Step	Action										
6	<p>Add an IMultiCellCanvas* part in the IFrameWindow* part and configure it as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Number of rows: 5</li> <li><input type="checkbox"/> Number of columns: 3</li> <li><input type="checkbox"/> Expandable rows: all</li> <li><input type="checkbox"/> Expandable columns: all</li> </ul>										
7	<p>Add three IMultiCellCanvas* parts in the IMultiCellCanvas* part already in place. The first multicell canvas contains the application title; the second, one graphic push button; and the third, a set canvas with the graphic push buttons and their associated static text controls.</p>										
8	<p>Configure each IMultiCellCanvas* part as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> IMultiCellCanvas* part <b>A</b> <ul style="list-style-type: none"> <li>— Number of rows: 3</li> <li>— Number of columns: 3</li> <li>— Expandable rows: Nos. 1, 3</li> <li>— Expandable columns: Nos. 1, 3</li> </ul> </li> <li><input type="checkbox"/> IMultiCellCanvas* part <b>B</b> <ul style="list-style-type: none"> <li>— Number of rows: 3</li> <li>— Number of columns: No. 3</li> <li>— Expandable rows: Nos. 1, 3</li> <li>— Expandable columns: 1, 3</li> </ul> </li> <li><input type="checkbox"/> IMultiCellCanvas* part <b>C</b> <ul style="list-style-type: none"> <li>— Number of rows: 3</li> <li>— Number of columns: 3</li> <li>— Expandable rows: Nos. 1, 3</li> <li>— Expandable columns: Nos. 1, 3</li> </ul> </li> </ul>										
9	<p>Add an IStaticText* part to cell (2, 2) of the IMultiCellCanvas* part, <b>A</b>, and change its <i>text</i> attribute to <b>Visual Realty Application</b>. This IStaticText* part constitutes the main title.</p>										
10	<p>Change the font of the main title to suit your needs. (In Figure 78 on page 221, the font is set to Times New Roman Bold, size 18. The resolution is SVGA.)</p>										
11	<p>Add an ISetCanvas* part to cell (2, 2) of the IMultiCellCanvas* part, <b>C</b>, and change the setting of its attributes as follows:</p> <table> <thead> <tr> <th>Attribute</th><th>Setting</th></tr> </thead> <tbody> <tr> <td>Pack Type</td><td>Even</td></tr> <tr> <td>Deck Count</td><td>2</td></tr> <tr> <td>Pad Width</td><td>20</td></tr> <tr> <td>Alignment</td><td>Center</td></tr> </tbody> </table>	Attribute	Setting	Pack Type	Even	Deck Count	2	Pad Width	20	Alignment	Center
Attribute	Setting										
Pack Type	Even										
Deck Count	2										
Pad Width	20										
Alignment	Center										

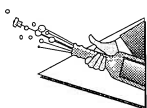
**Table 27.** (Part 3 of 3) Building ARealMainView

Step	Action																														
12	Add five IGraphicPushButton* parts: one to cell (2, 2) of the IMulti-CellCanvas* part, <b>E</b> , and four to the ISetCanvas* part.																														
13	<p>Change the IGraphicPushButton* parts settings as follows:</p> <table><tr><th>Button</th><th>Subpart</th><th>DLL</th><th>Res. id</th><th>Enabled</th></tr><tr><td>Logon</td><td>GPushButtonLogon</td><td>realicon</td><td>506</td><td>deselect</td></tr><tr><td>Properties</td><td>GPushButtonProperty</td><td>realicon</td><td>510</td><td>deselect</td></tr><tr><td>Buyers</td><td>GPushButtonBuyer</td><td>realicon</td><td>501</td><td>deselect</td></tr><tr><td>Sales</td><td>GPushButtonSale</td><td>realicon</td><td>509</td><td>deselect</td></tr><tr><td>Services</td><td>GPushButtonService</td><td>realicon</td><td>504</td><td>deselect</td></tr></table> <p>The last four graphic push buttons must be set to disable by deselecting the <b>Enabled</b> check box on the Control page of their settings notebook. This prevents the user from accessing a subsystem as long as no database connection is established (see “Logging on to the Database” on page 337). When a database connection is established, these graphic push buttons are set for <b>enable</b>.</p>	Button	Subpart	DLL	Res. id	Enabled	Logon	GPushButtonLogon	realicon	506	deselect	Properties	GPushButtonProperty	realicon	510	deselect	Buyers	GPushButtonBuyer	realicon	501	deselect	Sales	GPushButtonSale	realicon	509	deselect	Services	GPushButtonService	realicon	504	deselect
Button	Subpart	DLL	Res. id	Enabled																											
Logon	GPushButtonLogon	realicon	506	deselect																											
Properties	GPushButtonProperty	realicon	510	deselect																											
Buyers	GPushButtonBuyer	realicon	501	deselect																											
Sales	GPushButtonSale	realicon	509	deselect																											
Services	GPushButtonService	realicon	504	deselect																											
14	Add four IStaticText* parts to the ISetCanvas* part and change their label attributes as shown in Figure 78 on page 221.																														
15	Set the graphic push button logon as the default push button.																														
16	<p>Set the Tabbing and Depth Order and define two tabbing groups for the graphic push buttons as follows:</p> <table><tr><th>Group</th><th>Tab</th><th>Feature</th></tr><tr><td>X</td><td>X</td><td>GPushButtonLogon</td></tr><tr><td>X</td><td>X</td><td>GPushButtonProperty</td></tr><tr><td>-</td><td>X</td><td>GPushButtonBuyer</td></tr><tr><td>-</td><td>X</td><td>GPushButtonSale</td></tr><tr><td>-</td><td>X</td><td>GPushButtonService</td></tr></table> <p>Close the dialog box.</p>	Group	Tab	Feature	X	X	GPushButtonLogon	X	X	GPushButtonProperty	-	X	GPushButtonBuyer	-	X	GPushButtonSale	-	X	GPushButtonService												
Group	Tab	Feature																													
X	X	GPushButtonLogon																													
X	X	GPushButtonProperty																													
-	X	GPushButtonBuyer																													
-	X	GPushButtonSale																													
-	X	GPushButtonService																													
<b>Note:</b> Reverse highlighted letters are keyed to Figure 78 on page 221.																															

Now you can save ARealMainView. Switch to the Class Editor and fill in the *Code generation file* group box as follows:

- ☐ C++ header file (.hpp): **vrmain.hpp**
- ☐ C++ code file (.cpp): **vrmain.cpp**

Then, save the part.



Congratulations! You have finished building your visual parts. In Chapter 8, “Creating Nonvisual Parts,” on page 225 you will learn how to build nonvisual parts. Remember that visual programming does not imply no programming at all!



# 8

## Creating Nonvisual Parts

Nonvisual parts represent application objects that the user cannot see at run time. These objects can be divided into two main categories: business objects and technical objects.

Business objects support the business object model of the application. For example, `Property` is a business class whose instances are business objects. You can access the views that display some of their features (`APropertyView` displays the attribute values of `Property`).

Technical objects support the middleware layer of the application. They can be, for example, transaction management objects, security management objects, or database access objects. For the sample application, all Data Access Builder objects can be considered as technical objects. They provide the application with database access. Sometimes, technical objects can be used for more basic tasks, such as validating user input or handling the appearance of an object according to its contents. Event handlers are an example of such technical objects.

Basically you can handle two kinds of nonvisual parts with Visual Builder: the full, enabled nonvisual part and the class interface part. The full, enabled nonvisual part is always a subclass of the `IStandardNotifier` class, which provides a concrete implementation of the notifier protocol in the notification framework (see Chapter 10, “More about Visual Builder...,” on page 353). This part can notify other parts, and connections can be drawn from and to it.

The class interface part can inherit from any class but does not support the notification protocol. This nonvisual part cannot notify other parts. Creating a class interface part rather than a nonvisual part is especially useful when you do not want to alter the legacy source code of your own C++ classes that you want to use with Visual Builder. (See Chapter 10, “More about Visual Builder...,” on page 353 for more information about using your legacy code with Visual Builder.) However, creating a class interface part has two restrictions: Because you do not integrate any code to enable the notification framework, you cannot use a class interface part as a source for a connection. Also, if a class interface part is the target for an attribute-to-attribute connection, the connection is unidirectional (only the target side is updated). For a complete overview of the notification framework and more information about how to implement a nonvisual part, refer to Chapter 10, “More about Visual Builder...,” on page 353.

In the sections that follow, we describe how to build both full, enabled nonvisual parts and class interface parts. First you implement the `AMarketingInfo` nonvisual part, which belongs to the business object category. It holds business rules that apply to the properties in the context of our application. Then you implement a class interface part, `AMortgageCalculator`, which is used to estimate the highest mortgage the buyer can contract according to his financial profile. At last, you build the `NumOnlyKbdHandler` and `NumDecOnlyKbdHandler` event handlers that you use to force the user to enter numerical information (with or without decimal point) in the `ASimulMortgageView` visual part (for more information about `ASimulMortgageView` refer to “`ASimulMortgageView`” on page 187).

## AMarketingInfo

The `AMarketingInfo` part (see the design object model in Figure 40 on page 100) contains some attributes that are not stored in the database; *PricePerSqft* (price per square foot) is one of them. It is calculated from the *Price* and *Size* attributes of the Data Access Builder part, `Marketing_info`, by using the following formula:  $\text{PricePerSqft} = \text{Price} / \text{Size}$ . These attributes are called *derived* attributes because their value can be derived from the values of other attributes.

You can calculate the value of such attributes in two ways:

- ❑ Extend the `AMarketingInfo` nonvisual part by subclassing it. In effect, the source code that Data Access Builder generates is not intended to be edited directly. You lose all of your changes if you regenerate the code from the same class.
- ❑ Build a separate nonvisual part to hold the derived attributes and calculate their value from the `AMarketingInfo` part.

With the second way of calculating derived attributes, you do not have to associate the business rules with any implementation choice; that is, you do not have to store the information in a relational database such as DB2 for Windows. This is the calculation we choose. Building a nonvisual part constitutes an alternative to the custom logic connection that we describe later (see “Using Custom Logic” on page 271).

## Defining the Class Interface

The `AMarketingInfo` part consists of seven attributes and their associated get and set methods:

- ❑ *price*, the property price from the Data Access Builder part
- ❑ *commissionRate*, the commission rate from the Data Access Builder part (percentage of the total sale)
- ❑ *downPaymentRate*, the deposit rate the buyer has to pay to hold the property (percentage of the total price)
- ❑ *size*, the property size from the Data Access Builder part
- ❑ *commission*, the amount of commission the agent earns in selling the property
- ❑ *pricePerSqft*, the property price per square foot
- ❑ *downPayment*, the deposit the buyer has to pay to hold the property

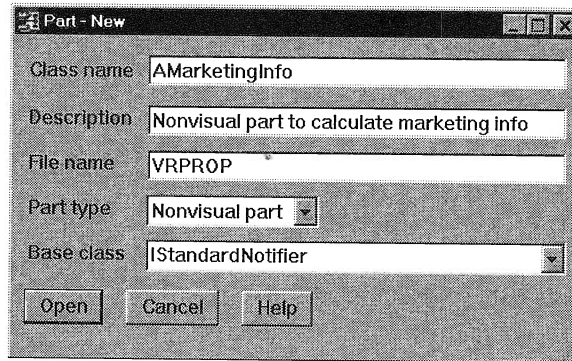
The first three attributes are updated according to the value of the last four attributes. In addition, each time one of these attributes is updated, all of the others are updated accordingly.

To create the `AMarketingInfo` nonvisual part, assuming Visual Builder has been started from the Property project folder:

1. Select the **New** option from the *Part* pull-down menu.
2. Fill in the *Class Name*, *Description*, and *File name* fields, as shown in Figure 79.
3. Select **Nonvisual part** in the *Part type* field.



- Click on the **Open** push button; the Part Interface Editor is loaded for editing the part.



**Figure 79.** Creating AMarketingInfo Nonvisual Part

To add *pricePerSqft* as a new attribute for the nonvisual part:

- Enter the attribute name in the corresponding field.
- Select **double** as the attribute type.
- Click on the **Defaults** push button. The different fields are filled in with the default member function prototypes for this attribute. Because the value of this attribute is updated by the set member function of the price attribute, you can erase the set member function of *pricePerSqft*.
- Fill in the *Description* field with a meaningful comment.
- Click on the **Add** push button.

You have now added the *pricePerSqft* attribute to the part. You can repeat the process for the *commission* and *downPayment* attributes. Refer to Table 28 for more information.

Because the other attributes have a set member function, you can add them by using the default member function prototypes. Therefore, you can directly click on the **Add with defaults** push button.

**Table 28.** (Part 1 of 2) AMarketingInfo Attributes

Name	Type	Get Function	Set Function	Event ID
price	double	virtual double price() const	virtual AMarketing- Info& setPrice(double aPrice)	priceId

**Table 28.** (Part 2 of 2) AMarketingInfo Attributes

Name	Type	Get Function	Set Function	Event ID
commission-Rate	double	virtual double commissionRate() const	virtual AMarketing-Info& setCommissionRate(double aCommissionRate)	commission-RateId
downPayment-Rate	double	virtual double downPaymentRate() const	virtual AMarketing-Info& setDownPaymentRate(double aDownPaymentRate)	down-Payment-RateId
size	double	virtual double size() const	virtual AMarketing-Info& setSize(double aSize)	sizeId
pricePerSqft	double	virtual double pricePerSqft() const		pricePer-SqftId
commission	double	virtual double commission() const		commissionId
downPayment	double	virtual double downPayment() const		down-PaymentId

Once you have registered all attributes and their associated methods, switch to the Class Editor and fill in the user file fields:

- ☐ User .hvp file: **vrpmrkt.hvp**
- ☐ User .cpv file: **vrpmrkt.cpv**

You are ready to generate the code:

1. Select *Save and Generate* → **Part source** to generate the part source code.
2. Select *Save and Generate* → **Feature source...** to generate the code for every get and set method registered.

Several files are generated:

- ☐ `vrpmrkt.h`, constant declaration for the class
- ☐ `vrpmrkt.hpp`, class declaration (this file includes `vrpmrkt.hpv`)
- ☐ `vrpmrkt.cpp`, class definition (this file includes `vrpmrkt.cpv`)
- ☐ `vrpmrkt.hpv`, attributes and member functions declaration
- ☐ `vrpmrkt.cpv`, member functions definition
- ☐ `vrpmrkt.rci`, resource file

You need to refine the accessor member functions to complete your nonvisual part.

## Refining the Feature Source Code

To refine the part and update the `pricePerSqft`, `commission`, and `downPayment` attributes according to the value of the other attributes, you must edit the `vrpmrkt.cpv` file. Use the LPEX editor, provided with VisualAge for C++, or your favorite editor to update the file, as shown in Figure 80.

```

// Feature source code generation begins here...
//.....
AMarketingInfo& AMarketingInfo::setPrice(double aPrice)
{
    if (!(iPrice == aPrice))
    {
        iPrice = aPrice;
        if( iSize != 0 ) {
            iPricePerSqft = IString(iPrice / iSize);
            iCommission = IString( iCommissionRate * iPrice / 100);
            iDownPayment = IString( iDownPaymentRate * iPrice / 100);
        }
        else
            iPricePerSqft = "Size is not informed.";
        notifyObservers(INotificationEvent(AMarketingInfo::priceId, *this));
        notifyObservers(INotificationEvent(AMarketingInfo::pricePerSqftId,
                                           *this));
        notifyObservers(INotificationEvent(AMarketingInfo::commissionId,
                                           *this));
        notifyObservers(INotificationEvent(AMarketingInfo::downPaymentId,
                                           *this));
    } // endif
    return *this;
}

//.....
AMarketingInfo& AMarketingInfo::setCommissionRate(double aCommissionRate)
{
    if (!(iCommissionRate == aCommissionRate))
    {
        iCommissionRate = aCommissionRate;
        if( iPrice != 0 )
            iCommission = IString( iCommissionRate * iPrice / 100);
        else
            iCommission = "Price is not informed.";
        notifyObservers(INotificationEvent(AMarketingInfo::commissionRateId,
                                           *this));
        notifyObservers(INotificationEvent(AMarketingInfo::commissionId,
                                           *this));
    } // endif
    return *this;
}

//.....
AMarketingInfo& AMarketingInfo::setDownPaymentRate(double aDownPaymentRate)
{
    if (!(iDownPaymentRate == aDownPaymentRate))
    {
        iDownPaymentRate = aDownPaymentRate;
        if( iPrice != 0 )
            iDownPayment = IString( iDownPaymentRate * iPrice / 100);
    }
}

```

**Figure 80.** (Part 1 of 2) MarketingInfo Source Code Detail

```

else
    iDownPayment = "Price is not informed";
    notifyObservers(INotificationEvent(AMarketingInfo::downPaymentRateId,
                                      *this));
} // endif
return *this;
}

//.....
AMarketingInfo& AMarketingInfo::setSize(double aSize)
{
    if (!(iSize == aSize))
    {
        iSize = aSize;
        if( iPrice != 0 )
            iPricePerSqft = IString(iPrice / iSize);
        else
            iPricePerSqft = "Price is not informed.";
        notifyObservers(INotificationEvent(AMarketingInfo::sizeId, *this));
        notifyObservers(INotificationEvent(AMarketingInfo::pricePerSqftId,
                                          *this));
    } // endif
    return *this;
}

// Feature source code generation ends here.

```

**Figure 80.** (Part 2 of 2) MarketingInfo Source Code Detail

Only the set member function for the *price*, *size*, *downPaymentRate*, and *commissionRate* attributes are updated to calculate the other attributes that depend on them. In “Connecting a Nonvisual Part to a Visual Part” on page 249, the MarketingInfo part is connected, by attribute-to-attribute connections, to the Marketing page of APropertyView. The contents of each entry field on this page are updated according to the value of the associated attribute. To keep the contents of the entry fields up to date, a notification must be sent to the attribute-to-attribute connections each time the value of an attribute changes. Considering a connection as an *observer*, the notification is triggered by the notifyObservers() method, which you must call in the get method of each attribute (see Chapter 10, “More about Visual Builder...,” on page 353). The notifications enable a global change on the Marketing page of APropertyView when necessary.

## AMortgageCalculator

AMortgageCalculator estimates the highest mortgage the potential buyer can contract based on his annual income, a given term in years and a given annual interest rate. Arbitrarily, we decide that no more than 30% of the annual income should be invested in the mortgage. The formula used can be expressed as follows:

$$\text{mortgage} = \text{income} \frac{30}{100} \exp\left(\frac{\text{rate}}{100} \text{years}\right)$$

The part is defined as a class interface since it is reused in Chapter 12, “More about SOM...,” on page 421 to illustrate how you can use the Direct-to-SOM feature with Visual Builder. The class interface is defined as shown in Table 29 and Table 30.

**Table 29.** AMortgageCalculator Attributes

Name	Type	Get Function	Set Function
income	double	double getIncome() const	void setIncome(double aIncome)
years	double	double getYears() const	void setYears(double aYears)
rate	double	double getRate() const	void setRate(double aRate)
estimation	double	double estimation() const	void setEstimation(double aEstimation)

**Table 30.** AMortgageCalculator Function

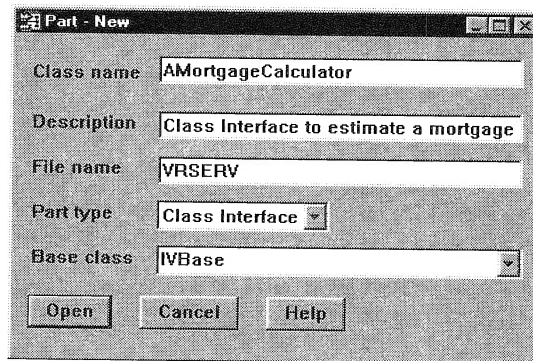
Name	Return Type	Parameters
estimate	double	N/A

Since AMortgageCalculator is a class interface it does not have the ability to notify other parts when its state changes. For this reason, no events are defined. In the following section we show you one way to build your class interface from scratch. In the section “Event Handler” on page 237, you will see how to create a class interface using an export file (VBE file).

## Defining the Class Interface

To create AMortgageCalculator, assuming Visual Builder has been started from the Common project folder:

1. Select the **New** option from the *Part* pull-down menu.
2. Fill in the *Class Name*, *Description*, and *File name* fields, as shown in Figure 81.
3. Select **Class Interface part** in the *Part type* field.
4. Click on the **Open** push button; the Part Interface Editor is loaded for editing the part.



**Figure 81.** Creating AMortgageCalculator Class Interface

Add the four attributes income, years, rate and estimation as shown previously for AMarketingInfo.

To add the estimate member function:

1. Click the **Action** tab of the Part Interface Editor to switch to the Action page.
2. Enter **estimate** in the *Action name* entry field.
3. Enter **estimate()** in the *Action member function* entry field (you specify this way that estimate does not require any parameter).
4. Enter **double** in the *Return type* entry field.
5. Enter a meaningful description of the estimate function in the *Description* entry field.
6. Click the **Add** push button to register your member function.

Once your class interface is defined, switch to the Class Editor by clicking on the left pie in the lower right corner of your window and, in the *Code generation files* group box, fill in the *C++ header file (.hpp)* and *C++ code file (.cpp)* entry fields with the respective values: **vrssimc.hpp** and **vrssimc.cpp**.

Save your part by selecting the File Save menu item from the main menu. Notice that the *Save and Generate* option is disabled since you must provide the code! This is what the next section is all about.

## Writing the Source Code

From the Common project, start the LPEX editor and type in the header file of AMortgageCalculator as shown in Figure 82.

```
#ifndef _AMORTGAGECALCULATOR_
#define _AMORTGAGECALCULATOR_
//*****
//
// Simple Mortgage Calculator
//
//*****

class AMortgageCalculator
{
public:
    AMortgageCalculator();
    ~AMortgageCalculator();

    double income;
    double years;
    double rate;
    double estimation;

    double getIncome();
    double getYears();
    double getRate();
    double getEstimation();
    void setIncome(double aIncome);
    void setYears(double aYears);
    void setRate(double aRate);
    void setEstimation(double estimation);
};
#endif
```

**Figure 82.** AMortgageCalculator Header File: vrssimc.hpp

Save the file. Create another file for AMortgageCalculator as shown in Figure 83.



```
#include "vrssimc.hpp"
#include "math.h"

AMortgageCalculator::AMortgageCalculator() {}

AMortgageCalculator::~AMortgageCalculator() {}

void AMortgageCalculator::setYears(double y)
{
    years = y;
}

void AMortgageCalculator::setRate(double r)
{
    rate = r;
}

void AMortgageCalculator::setIncome(double i)
{
    income = i;
}

void AMortgageCalculator::setEstimation(double e)
{
    estimation = e;
}

double AMortgageCalculator::getYears()
{
    return years;
}

double AMortgageCalculator::getIncome()
{
    return income;
}

double AMortgageCalculator::getEstimation()
{
    return estimation;
}

double AMortgageCalculator::getRate()
{
    return rate;
}

double AMortgageCalculator::Estimate()
{
    double result = income*30.0/100*exp(rate/100*years);
    setEstimation(result);
    return result;
}
```

**Figure 83.** AMortgageCalculator Implementation File: vrssimc.cpp

Save the implementation file. Your part is completed.

## Event Handler

The parts you can create with Visual Builder provide standard behaviors that should meet most of your needs. At times, however, you may want to use an event handler to modify or extend the default behavior.

Below, we describe how to build two simple event handlers to filter the characters the user types in ASimulMortgageView. These handlers are implemented as class interface parts in order to reuse them with Visual Builder. For more information about building event handlers, refer to the *Open Class Library User's Guide*.

Handlers are registered with parts and set up in the Handlers pages of the settings notebook. You can attach several handlers to the same part. They will be executed in the reverse order in which they are attached to the part.

The management of events and event handlers is based on the IBM Open Class Library architecture. Basically, the architecture uses events and handlers to encapsulate the message architectures of Windows, OS/2 PM, and Motif in an object-oriented way. Windows messages are sent as event objects to the window or control that received the event. The window then invokes the handler attached to it, passing the event object as a parameter. The handlers are called sequentially, with the most recently added handler called first. When an event handler completes the event processing, it returns a Boolean value of TRUE. If none of the handlers can process the event, it is passed up to Windows.

The distinction between window classes and handler classes lets you separate the event-handling logic from the rest of the application. Thus, you can reuse the logic in several applications. For example, you can build an event handler that verifies the social security number format whenever an entry field accepts the social security number. You define this handler only once. Any time you want to use an entry field to enter the social security number information, you can attach the event handler to the entry field.

You can use the IHandler part to control the event handler action on a specific part.

You build an event handler in two steps:

1. Write the code of your event handler class:

- Derive your event handler class from one of the predefined event handler classes. If none of them meets your needs, you still can derive your handler from the `IHandler` part.
  - Each handler class has one or more virtual functions that are called to process the events. You provide your own function to override these virtual functions and tailor its logic to your needs.
  - Compile the new event handler class and create a library that you will use later on when linking the whole application.
2. Convert your event handler class to a class interface part that you can use with Visual Builder.

## Writing the Code for Your Event Handler Class

To write the code for your `NumOnlyKbdHandler` and `NumDecOnlyKbdHandler` classes, you first have to choose the handler class from which to derive your own event handler. Because the event handler is to be used when entering information from the keyboard, it seems natural to choose `IKeyboardHandler` as the class from which to derive your own event handler. Of course, you also could choose an event handler associated with the editing process, such as `IEditHandler`, but you would not be able to reuse it with other types of controls, such as a list box. In our example, you derive both handler classes from `IKeyboardHandler` (Figure 84), and you overload the virtual function, `characterKeyPress`, to meet your needs (Figure 85).

```

#ifndef _KBDHDR_
#define _KBDHDR_
//*****
// Reusable Handlers - Keyboard Handler
//
// Copyright (C) 1994, Law, Leong, Love, Olson, Tsuji.
//
// Adapted for use with VisualAge C++ by: Peter Jakab
// Adapted for Windows by: Michael Friess
//
// All Rights Reserved.
//*****
#include <ikeyhdr.hpp>
#include <istring.hpp>
#include <iwindow.hpp>
#ifndef __NO_DEFAULT_LIBS__
    #pragma library("kbdhdr.lib")
#endif

// Keyboard handler example to restrict character input to
// digits ( 0-9 )
class NumOnlyKbdHandler : public IKeyboardHandler
{
protected:
    virtual Boolean characterKeyPress ( IKeyboardEvent& event );
};

// Keyboard handler example to restrict character input to
// digits ( 0-9 ) and ( ., )
class NumDecOnlyKbdHandler : public IKeyboardHandler
{
protected:
    virtual Boolean characterKeyPress ( IKeyboardEvent& event );
};

#endif                                     /* _KBDHDR_ */

```

**Figure 84.** NumOnlyKbdHandler and NumDecOnlyKbdHandler Header File

To implement NumOnlyKbdHandler, you overload the characterKeyPress function to filter the characters typed from the keyboard and ensure they are only numbers (see Figure 85).

```

#if __OS2__
#define INCL_WINDIALOGS                /* For WinAlarm.                */
#define INCL_WININPUT                  /* For KC_CHAR, VK_DELETE.      */
#include <os2.h>
#elif __WINDOWS__
#include <windows.h>                    /* MessageBeep, VK_DELETE      */
#ifndef __NO_DEFAULT_LIBS__
#pragma library("user32.lib")
#endif
#endif
#include "kbdhdr.hpp"
Boolean NumOnlyKbdHandler :: characterKeyPress ( IKeyboardEvent& event )
{
    Boolean badKey = true;                /* Assume an invalid key,*/
    Boolean dontPassOn = true;
    event.setResult(true);                /* (Ignored if return false.)*/
    IString strChar = event.mixedCharacter();
    if (strChar.isSBCS())
    {
        /* Single-byte input.*/
        if (strChar.isDigits())
        {
            /* '0'-'9'*/
            badKey = false;                /* Valid digit.*/
            dontPassOn = false;            /* Pass event to the window.*/
        }
        else if (strChar == " ")
        {
            /* Space bar.*/
            badKey = false;                /* Replace with Delete key.*/
            #if __OS2__
                IEventParameter1
                    param1( event.parameter1().number1() & ~KC_CHAR,
                            event.parameter1().char3(), 0 );
                IEventParameter2 param2( 0, VK_DELETE);
            #elif __WINDOWS__
                IEventParameter1 param1( VK_DELETE);
                IEventParameter2 param2( event.parameter2());
            #endif
            event.window()->sendEvent( IWindow::character, param1, param2);
            event.setResult( true );
        }
        // Throw away any other character key.
    }
    /* End single-byte input.*/
    if (badKey)
    {
        #if __OS2__
            WinAlarm( IWindow::desktopWindow()->handle(), WA_WARNING );
        #elif __WINDOWS__
            MessageBeep( MB_ICONEXCLAMATION);
        #endif
    }
    return dontPassOn;
}

```

**Figure 85.** NumOnlyKBDHandler Definition

To implement NumDecOnlyKbdHandler, you overload the character-KeyPress function to filter the characters typed from the keyboard and ensure they are only numbers or a decimal point (see Figure 86).

```

#if __OS2__
#define INCL_WINDIALOGS                /* For WinAlarm.                */
#define INCL_WININPUT                  /* For KC_CHAR, VK_DELETE.      */
#include <os2.h>
#elif __WINDOWS__
#include <windows.h>                  /* MessageBeep, VK_DELETE      */
#ifndef __NO_DEFAULT_LIBS__
#pragma library("user32.lib")
#endif
#endif
#include "kbdhdr.hpp"
Boolean NumDecOnlyKbdHandler :: characterKeyPress ( IKeyboardEvent& event )
{
    Boolean badKey = true;                /* Assume an invalid key,*/
    Boolean dontPassOn = true;
    event.setResult(true);                /* (Ignored if return false.)*
    IString strChar = event.mixedCharacter();
    if (strChar.isSBCS())
    {
        /* Single-byte input.*/
        if (( strChar.isDigits() ) || ( strChar.indexOfAnyOf( "." ) ))
        {
            /* '0'-'9'.,'*/*
            badKey = false;                /* Valid digit.*/
            dontPassOn = false;            /* Pass event to the window.*/
        }
        else if (strChar == " ")
        {
            /* Space bar.*/
            badKey = false;                /* Replace with Delete key.*/
            #if __OS2__
                IEventParameter1
                    param1( event.parameter1().number1() & ~KC_CHAR,
                        event.parameter1().char3(), 0 );
                IEventParameter2 param2( 0, VK_DELETE);
            #elif __WINDOWS__
                IEventParameter1 param1( 0, VK_DELETE);
                IEventParameter2 param2( event.parameter2());
            #endif
            event.window()->sendEvent( IWindow::character, param1, param2);
            event.setResult( true );
        }
        // Throw away any other character key.
    }
    /* End single-byte input.*/
    if (badKey)
    {
        #if __OS2__
            WinAlarm( IWindow::desktopWindow()->handle(), WA_WARNING );
        #elif __WINDOWS__
            MessageBeep( MB_ICONEXCLAMATION);
        #endif
    }
    return dontPassOn;
}

```

**Figure 86.** NumDecOnlyKbdHandler Definition

Notice that the implementation code for both handlers is also compatible with the OS/2 IBM Open Class Library.

Once the handlers are implemented, you need to define their interface by writing a Visual Builder export file (VBE). This file is used to import the class interface of your handlers in Visual Builder and create a VBB file that contains the two handlers.

## Creating a Class Interface Part from Your Event Handler Class

To create the NumOnlyKbdHandler and NumDecOnlyKbdHandler class interface parts, you must create the definition of each part in Visual Builder. Actually, creating a keyboard handler part in Visual Builder is very simple, because a keyboard handler does not have any features. Visual Builder must have the following information to use a keyboard handler:

- ❑ Name of the header file where the keyboard handler class is declared, namely, `kbdhdr.hpp`
- ❑ Macro defined in the header file to avoid multiple inclusions, namely, `_KBDHDR_`
- ❑ Name of the library where the definition of the keyboard handler definition is stored. The library name is used at link time. The simplest way of handling the library is to declare it in the source code with the `#pragma library` statement:

```
#ifndef __NO_DEFAULT_LIBS__
    #pragma library("kbdhdr.lib")
#endif
```

The linker uses this information to retrieve the name of the library it needs to build the application.

If you do not use the `#pragma library` statement, you can specify the name of the library in the linking options of the project where you are using the keyboard handler.

The simplest and fastest way of creating the class interface part for each handler is to write a Visual Builder export file (or VBE file) and import it into Visual Builder. A VBE file is a flat file that you use to describe a part and its interface. Use your favorite editor to create the `KBDHDR.VBE` file with the following contents:

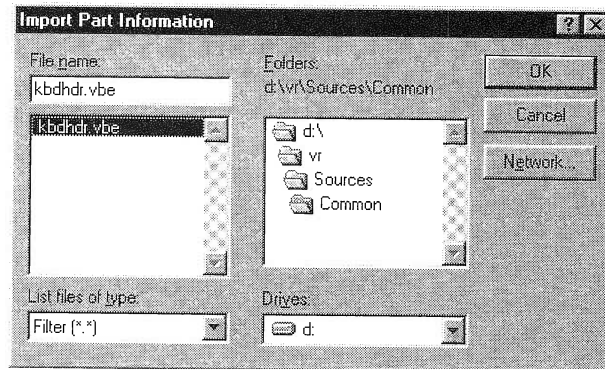
```
//VBBeginPartInfo: NumOnlyKbdHandler
//VBParent: IKeyboardHandler
//VBIncludes: "kbdhdr.hpp" _KBDHDR_
//VBPartDataFile: KBDHDR.VBB
//VBComposerInfo: class,204,dde4vr30
//VBPreferredFeatures: this
//VPEndPartInfo: NumDecOnlyKbdHandler
```



```
//VBBeginPartInfo: NumOnlyKbdHandler
//VBParent: IKeyboardHandler
//VBIncludes: "kbdhdr.hpp" _KBDHDR_
//VBPartDataFile: KBDHDR.VBB
//VBComposerInfo: class,204,dde4vr30
//VBPreferredFeatures: this
//VPEndPartInfo: NumDecOnlyKbdHandler
```

You can declare as many class interfaces as needed in your VBE file as long as you delimit them with the `//VBBeginPartInfo` and `//VPEndPartInfo` tags.

To import the `KBDHDR.VBE` file in Visual Builder, start Visual Builder and select the **File→Import Part Information...** menu item. Enter the information as shown in Figure 87.



**Figure 87.** Visual Builder: Importing kbdhdr.vbe Part Information File

Visual Builder creates a VBB file (`kbdhdr.vbb`) from the information stored in the VBE file. You are now ready to use your event handlers.

In the section that follow we use the `NumOnlyKbdHandler` to illustrate how you can dynamically activate an event handler using Visual Builder.

## Using Your Keyboard Handler

Before using the event handler, you must import it as a nonvisual part in Visual Builder and open the settings of the part to which you want to attach the handler. Select the **Handlers** page and enter the name of the handler you want to attach to the part. If several handlers have already been attached to this part, you can choose the order in which

you want your handler to be added to the handler list with the **Add before** or **Add after** push buttons (see Figure 58 on page 165). The handlers are activated according to the order in the list.

You can use the nonvisual part, `IHandler`, to control the activation and deactivation of the handler.

Visual Builder provides you with an `IHandler` part to process a specific event of a part. The `IHandler` part attributes and actions are:

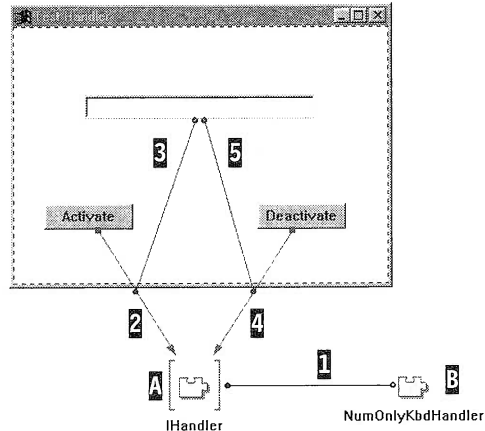
❑ **Attribute**

- *enabled*: indicates whether or not the handler is enabled

❑ **Actions**

- *disable*: disables the handler so that it does not process window events
- *start*: attaches the handler to the `IWindow` object
- *stop*: detaches the handler from the `IWindow` object

As an example, suppose you want to attach an event handler to an entry field or detach an event handler from an entry field when clicking on a specific push button (Figure 88):



**Figure 88.** Simple Application with Handler

1. Add an `IHandler` variable part to the free-form surface, **A**.
2. Add the event handler part you built to the free-form surface, **B**.
3. Connect the *this* of the event handler to the *this* of the variable, **1**.

4. Connect the **buttonClickEvent** event of the *Activate* push button to the **start** action of the IHandler variable part, **2**, and pass the **this** of the entry field to the connection as a parameter, **3**. In this way, when the push button is clicked, the event handler is attached to the entry field.
5. Connect the **buttonClickEvent** event of the *Deactivate* push button to the **stop** action of the IHandler variable part, **4**, and pass the **this** of the entry field to the connection as a parameter, **5**. In this way, when the push button is clicked, the event handler is detached from the entry field.

Save and generate the part. Compile and link your code (make sure that the `kbdhdr.lib` and `kbdhdr.hpp` files are located in your working directory).

To test your sample, type anything in the entry field and verify that nothing happens. Now activate the event handler by pressing the **Activate** push button. If you try to type anything else than a number a beep sounds. Now deactivate the event handler by pressing the **Deactivate** push button. You should be able to type anything in the entry field.

So far, you have built all of the parts that you need to make up the Property subsystem; that is, the parts that constitute the static structure of the subsystem. Now, you need some “glue,” so that the parts can communicate with each other; that is, you must build the dynamic structure of the subsystem by connecting the parts. That is what you do in Chapter 9, “Connecting the Parts,” on page 247.

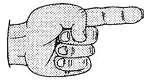
# 9

## Connecting the Parts

Before you begin this chapter, we suggest that you refer to “Using Visual Builder” on page 24 for a complete description of connections. From now on, we assume that you have enough knowledge to distinguish one connection from another.

To help you implement the connections between parts, we use the CRC cards, as well as the static and the dynamic model refined at the design level (see Chapter 4, “Designers at Work,” on page 81).

In this chapter, we show you how to connect the parts, and we explain the various connections. You refine each view in a bottom-up order according to the view hierarchy (Figure 50 on page 149). For each view, we provide step-by-step guidelines to help you add the necessary subparts and build the connections to implement the part’s logic. We strongly recommend that you name your subparts with the names that are shown in the figures in this chapter.

**Read This**

For every part to be refined, you must locate the corresponding WorkFrame project and open the associated VBB file—by double-clicking with your left mouse button—to start Visual Builder.

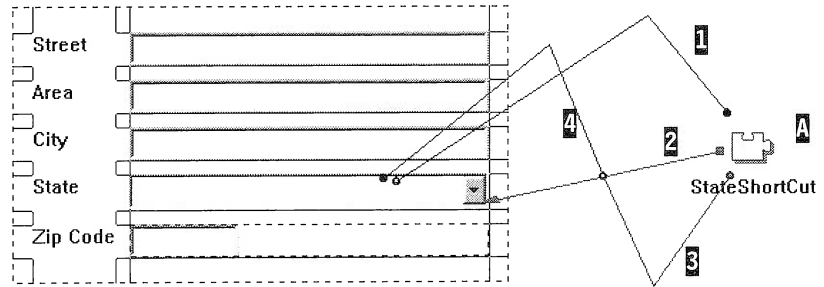
## AAddressView

Among the property information that AAddressView displays, the state appears in ComboStateBox as the state mnemonic concatenated with the state full name. In the PROPERTY\_ADDRESS table, the state is stored using the two-letter mnemonic. The translation from the string displayed in ComboStateBox to the two-letter string stored in the table is automatically handled by truncation of the string contained in the combination list box when the data is stored in the table. However, when a record is read from the table and displayed in AAddressView, the state displayed in ComboStateBox must match the mnemonic stored in the PROPERTY\_ADDRESS table for this record. Although this matching is automatically performed by the behavior of the OS/2 combination list box, you must provide, in Windows, a nonvisual part to locate the proper state string in a combination list box.

## Using Sample Parts

Along with the basic part library (VBBASE.VBB), the database access part library (VBDAX.VBB), and the multimedia part library (VBMM.VBB) provided with Visual Builder, comes another useful part library: VBSAMPLE.VBB. This library deserves to be called the *programmer toolbox* of Visual Builder. It includes parts that enhance the basic types of the C++ language, such as String, Long or Boolean. It also includes some general parts that you can reuse for basic file input/output or mathematical computation (see also “AUpLoadView” on page 324).

In this view, you use IVBStringPart to hold the state retrieved from the database and used to make the proper selection in the state combination list box of AAddressView (see Figure 89). This part contains many methods that you might find useful for your future applications.



**Figure 89.** AAddressView Connections

IVBStringPart is the equivalent version, for the Visual Builder, of the standard IString class. Unlike the class interface IString, IVBStringPart is a fully enabled part that can notify other parts. You will use it, for example, to hold the SQL clause from which the agent selects a list of properties in APropertySearchParameterView (see “Building the Clause” on page 311).

In the rest of the application, you will use two other parts from VBSAMPLE.VBB:

- ❑ **IVBLongPart** is the equivalent version, for the Visual Builder, of the standard *long* type. You will use it, for example, to hold the number of days a property is on the market. (See “Building the Clause” on page 311).
- ❑ **IVBBooleanPart** is the equivalent version, for the Visual Builder, of the standard *IBoolean* class. You will use it to build the SQL clause from check box controls in APropertySearchParameterView (See “Managing the User Input” on page 308).

## Connecting a Nonvisual Part to a Visual Part

Follow the steps in Table 31 to implement the selection of a state item in a combination list box and connect IVBStringPart to AAddressView.

Table 31. (Part 1 of 2) Implementing a State Selection	
Step	Action
1	Open the AAddressView part.

**Table 31.** (Part 2 of 2) Implementing a State Selection

Step	Action
2	Add an IVBStringPart* part, on the free-form surface, <b>A</b> . (IVBStringPart part is located in the VBSAMPLE.VBB file. Use the <b>Option</b> → <b>Add part...</b> from the Composition Editor pull-down menu to add it to the free-form surface.)
3	Connect the <b>text</b> attribute of the IVBStringPart* part to the <b>text</b> attribute of the <i>ComboBoxState</i> combination list box, <b>B</b> .  This attribute-to-attribute connection guarantees that the <b>text</b> attributes of the two parts are in sync.
4	Connect the <b>text</b> event of the IVBStringPart* part to the <b>locate-Text</b> action of the ComboBoxState, <b>C</b> .  When the <b>text</b> attribute of the IVBStringPart* part changes, the state is located within the state combination list box. This connection needs the string to be located as the parameter to execute.
5	Connect the <b>text</b> attribute of the IVBStringPart* part to the <b>searchString</b> parameter of the connection <b>C</b> (see connection <b>B</b> and “Passing a Parameter to a Connection Dynamically” for more details).
6	Connect the <b>actionResult</b> attribute from the connection <b>C</b> to the <b>selection</b> attribute of the ComboBoxState combination list box, <b>D</b> .
<b>Note:</b> Reverse highlighted numbers and letters are keyed to Figure 89 on page 249.	

You can now save your part and generate its code. From the Visual Builder window, select **Save and generate** → **Part source** in the *File* pull-down menu.

## Passing a Parameter to a Connection Dynamically

Sometimes an event-to-action connection needs one or several parameters to execute. In such cases, the connection displays as a dashed line (unless default parameter are provided). You can provide the connection with the parameters in two ways:

- ❑ Open the settings of the connection and click on the **Set Parameters...** push button. A dialog box is displayed and you are prompted to give the parameter’s value. If the parameter is a text string and the type of the parameter is an IString, you can enter the string without double quotes. Otherwise, you must surround the string with double quotes. In this case the parameters are set

statically: Each time the connection fires, it uses the same parameter values (see “Passing a Parameter to a Connection Statically” on page 271 for an example of static parameter).

- ❑ Connect an attribute of the matching type to the parameter of the connection. This parameter is shown in the connection pop-up menu. The parameter connection is a kind of attribute-to-attribute connection, but it is unidirectional (see “Using Visual Builder” on page 24). In this case, the parameters are set dynamically: Each time the connection fires, it uses the parameter values according to the parameter connection.

In AAddressView you use a parameter connection to dynamically provide the locateText action of the combination list box with the string to be searched. This string is provided by the contents of the IVBStringPart. In “Passing a Parameter to a Connection Statically” on page 271, you use the first method to provide a parameter to a connection statically.

## APropertyView

Four groups of connections are built in APropertyView:

- ❑ A group that enables the user to select a video file (Figure 90 on page 252)
- ❑ A group that enables the user to play a video file (Figure 90 on page 252)
- ❑ A group that enables the user to edit within the multiple-line edit (MLE) control of the description page (Figure 91 on page 255)
- ❑ A group that updates the marketing page according to the business logic stored in the AMarketingInfo nonvisual part (Figure 92 on page 257)

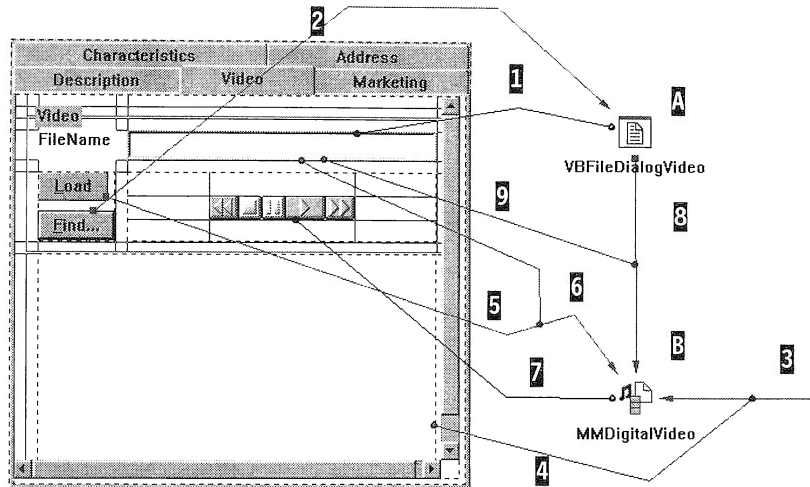
We explain these connections below.

### Selecting a Video File

In the Visual Realty application, the user selects a video file by clicking on the **Find...** push button (see Figure 90). Notice that, in “APropertyView” on page 160, the *FileName* entry field is set to read-only, so the user must click on **Find...** to select a file. This limiting condition prevents the user from mistyping the video file name, path name, or both.

Visual Builder provides the IVBFileDialog part for selecting files. It encapsulates the behavior of the standard file dialog window available in Windows NT or Windows 95.





**Figure 90.** Connections for Selecting a Video File

Follow the step-by-step instructions in Table 32 to implement the selection of a video file.

**Table 32.** (Part 1 of 2) Implementing a Video File Selection

Step	Action
1	Open the APropertyView part.
2	Switch to the Video page.
3	Add an IVBFileDialog* part on the free-form surface, <b>A</b> (IVBFileDialog part is located in the <i>Other</i> category).
4	Open the settings notebook of IVBFileDialog and set the values as follows: <div style="display: flex; justify-content: space-between;"> <div><b>Field</b></div> <div><b>Value</b></div> </div> <div style="display: flex; justify-content: space-between;"> <div>Title</div> <div>Digital Video</div> </div> <div style="display: flex; justify-content: space-between;"> <div>File name</div> <div>*.AVI</div> </div> <p>These values tailor the file dialog box and allow filtering of the files for the selection.</p>
5	Connect the <b>filename</b> attribute of the IVBFileDialog* part to the <b>text</b> attribute of the <i>VideoFileName</i> entry field, <b>1</b> . <p>This attribute-to-attribute connection guarantees that whenever the user selects a file, the content of the <i>VideoFileName</i> entry field is changed accordingly.</p>

**Table 32.** (Part 2 of 2) Implementing a Video File Selection

Step	Action
6	<p>Connect the <b>buttonClickEvent</b> event of the <i>Find...</i> push button to the <b>ShowModally</b> action of the IVBFileDialog* part, <b>2</b>.</p> <p>Use the <b>ShowModally</b> action to force the user to select a file name before returning to the Video page of the notebook.</p>
<p><b>Note:</b> Reverse highlighted numbers and letters are keyed to Figure 90 on page 252.</p>	

## Adding Multimedia Features

To enable the user to play digital video files, use the IMMDigitalVideo part from the \IBMCPWP\IVB\VBMM.VBB file. Among other nifty features, this part enables users to load a video file (extension .AVI) on a different thread and play it in the window of their choice.

Follow the step-by-step instructions in Table 33 to add multimedia features to your part.

**Table 33.** (Part 1 of 2) Implementing Multimedia Features

Step	Action
1	<p>Add an IMMDigitalVideo* part, on the free-form surface, <b>3</b>. (IMMDigitalVideo part is located in the VBMM.VBB file. Use the <b>Option → Add part...</b> from the Composition Editor pull-down menu to add it to the free-form surface.)</p>
2	<p>Connect the <b>ready</b> event of the APropertyView part to the <b>setWindow</b> action of the IMMDigitalVideo* part, <b>3</b>.</p> <p>This connection requires a parameter: the window where the video is played.</p>
3	<p>Connect (see <b>4</b>) the <b>handle</b> attribute of the ICanvas* part to the connection <b>3</b>.</p> <p>This connection enables the video to be played on the canvas. This connection is not necessary if you do not want to specify a window for the video. If you do not specify a window for the video, the video is played in a default window created by the part.</p>
4	<p>Connect the <b>buttonClickEvent</b> event of the <i>Load</i> push button to the <b>loadOnThread</b> action of the IMMDigitalVideo* part, <b>3</b>.</p> <p>The connection requires a video file name as a parameter (notice that the connection is dashed).</p>

**Table 33.** (Part 2 of 2) Implementing Multimedia Features

Step	Action
5	Connect (see <b>6</b> ) the <b>text</b> attribute of the <i>VideoFileName</i> entry field to the <b>filename</b> attribute of the connection <b>5</b> .
6	Connect the <b>playableDevice</b> attribute of the IMMPlyerPanel* part to the <b>this</b> attribute of the IMMDigitalVideo* part, <b>7</b> .  This connection associates each behavior feature of IMMDigitalVideo with the corresponding animated button of the IMMPlyerPanel.
7	Connect the <b>pressedOKEvent</b> event of the IVBFileDialog* part to the <b>loadOnThread</b> action of the IMMDigitalVideo* part, <b>8</b> .  This connection loads the video file name so that the video can be played.
8	Connect (see <b>9</b> ) the <b>text</b> attribute of the <i>VideoFileName</i> entry field to the <b>filename</b> parameter of the connection <b>8</b> .
<b>Note:</b> Reverse highlighted numbers and letters are keyed to Figure 90 on page 252.	

**Tip**

If you have trouble accessing the controls located in the video group box, you can move the group box to the free-form surface, make your connections, and move the group box back to the video page. To move the group box back to the video page, follow these steps:

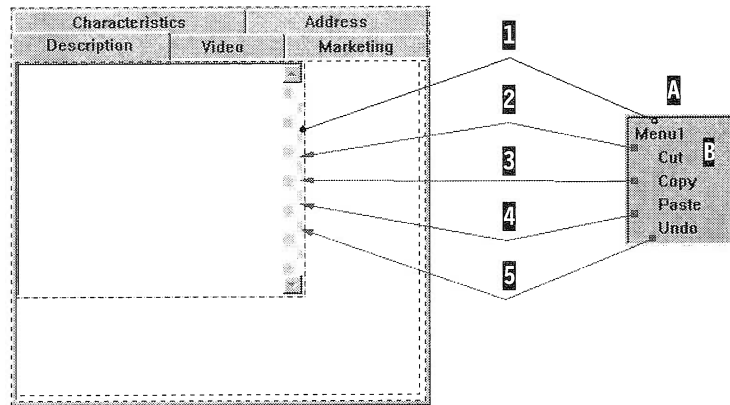
1. Resize it down on the free-form surface.
2. Drop it on one of the upper-left cells of the multicell canvas.
3. Resize it up with the mouse while holding the **ALT** key.

The **ALT** key lets you resize a control to make it span multiple cells of a multicell canvas.

You can also display the tabbing and depth order of your parts by selecting the **View Parts List...** option from the pop-up menu of any composer parts, and move the group box above the controls it surrounds.

## Adding a Pop-up Menu

To facilitate editing the description page, add a pop-up menu to the MLE control so that the user can use its copy, cut, paste, and undo functions (see Figure 91). For more information about how to build a menu, refer to the *Visual Builder User's Guide*.



**Figure 91.** Building a Pop-up Menu for the Multiple-Line Edit Control

To implement these connections, follow the steps in Table 34.

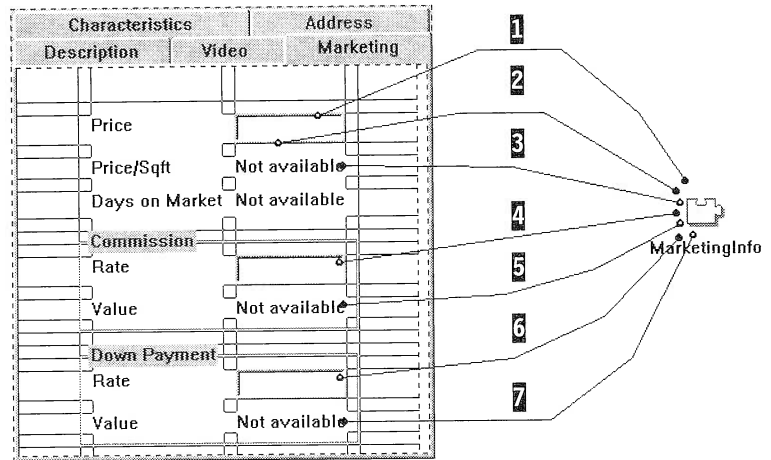
Table 34. (Part 1 of 2) Building a Pop-up Menu	
Step	Action
1	Switch to the Description page.
2	Add an IMenu* part on the free-form surface, <b>A</b> . (The IMenu part is located in the <i>Frame extensions</i> category.)
3	Click on the <b>Sticky</b> check box at the bottom of the palette. When you select <i>Sticky</i> , the mouse remains loaded with the part you last selected. This makes it easy to drop several copies of the same part.
4	Add four IMenuItem* parts on the part, <b>A</b> (see <b>B</b> for the first menu item).  Because the <i>Sticky</i> check box is selected, you do not have to reselect the IMenuItem* part from the parts palette.
5	Deselect the <b>Sticky</b> check box.
6	Add an IMenuSeparator* part between the third and fourth menu item.
7	Change the label of each menu item to match those in Figure 91.
8	Connect the <b>menu</b> attribute of MultiLineEditDescription to the <b>this</b> attribute of the IMenu* part, <b>A</b> .  The menu is associated with the MLE control.

Table 34. (Part 2 of 2) Building a Pop-up Menu	
Step	Action
9	Connect the <b>commandEvent</b> event of the <i>Cut</i> menu item to the <b>cut</b> action of MultiLineEditDescription, <b>2</b> .
10	Connect the <b>commandEvent</b> event of the <i>Copy</i> menu item to the <b>copy</b> action of MultiLineEditDescription, <b>3</b> .
11	Connect the <b>commandEvent</b> event of the <i>Paste</i> menu item to the <b>paste</b> action of MultiLineEditDescription, <b>4</b> .
12	<p>Connect the <b>commandEvent</b> event of the <i>Undo</i> menu item to the <b>undo</b> action of MultiLineEditDescription, <b>5</b>.</p> <p>You can access the <b>undo</b> action from a dialog box, which is displayed when you select the <i>More...</i> option in the MultiLineEditDescription pop-up menu.</p>
<b>Note:</b> Reverse highlighted numbers and letters are keyed to Figure 91 on page 255.	

## Connecting AMarketingInfo to the Marketing Page

To update the value of its different static text controls, the Marketing page must be able to access the services of the AMarketingInfo nonvisual part (refer to “AMarketingInfo” on page 226).

The entry fields of APropertyView provide AMarketingInfo with the values for the price, size, commission rate, and down payment rate. In return, the AMarketingInfo part provides APropertyView with the values for the price/sqft, commission, and down payment (Figure 92).



**Figure 92.** Connections between AMarketingInfo and Marketing Page

To implement these connections, follow the step-by-step instructions in Table 35.

Table 35. (Part 1 of 2) Using AMarketingInfo Part	
Step	Action
1	Switch to the Marketing page.
2	Add a AMarketingInfo* part on the free-form surface, <b>A</b> (use the <b>Option</b> → <b>Add part...</b> from the Composition Editor pull-down menu).
3	Connect the <i>valueAsDouble</i> attribute of EntryFieldPrice to the <i>price</i> attribute of AMarketingInfo*, <b>1</b> . Use the <i>valueAsDouble</i> attribute instead of the <i>text</i> attribute because the AMarketingInfo part expects a double value.
4	Connect the <i>valueAsDouble</i> attribute of EntryFieldSize to the <i>size</i> attribute of AMarketingInfo*, <b>2</b> . Note that the <i>size</i> attribute is located on the Characteristics page.
5	Connect the <i>pricePerSqft</i> attribute of AMarketingInfo* to the <i>text</i> attribute of staticTextPriceSqft, <b>3</b> .
6	Connect the <i>valueAsDouble</i> attribute of EntryFieldCommission-Rate to the <i>commissionRate</i> attribute of AMarketingInfo*, <b>4</b> .

**Table 35.** (Part 2 of 2) Using AMarketingInfo Part

Step	Action
7	Connect the <b>commissionValue</b> attribute of AMarketingInfo to the <b>text</b> attribute of staticTextCommissionValue, <b>5</b> .
8	Connect the <b>valueAsDouble</b> attribute of EntryFieldDownPaymentRate to the <b>downPaymentRate</b> attribute of AMarketingInfo, <b>6</b> .
9	Connect the <b>downPayment</b> attribute of AMarketingInfo to the <b>text</b> attribute of staticTextDownPaymentValue, <b>7</b> .
<b>Note:</b> Reverse highlighted numbers and letters are keyed to Figure 92. The static text control, <i>StaticTextDaysOnMarket</i> , is not updated in this view. It will be connected in another view.	

You can now save your part and generate its code. From the Visual Builder window, select **Save and generate** → **Part source** in the *File* pull-down menu.

## Design Considerations

In most cases, it is worthwhile to encapsulate service parts in other parts. Such encapsulation enhances reusability while reducing complexity. In APropertyView, you embed the multimedia features, file dialog window, and AMarketingInfo part to hide the complexity of the view and make it a self-contained part that is easily reusable. It is possible to build separate parts for the different pages and embed the nonvisual parts in the pages. However, because the pages are simple and will not be reused, they are directly implemented at the notebook level (see “Using a Notebook Control” on page 162).

At this point, you may wonder why you do not embed the Data Access Builder nonvisual parts and the database services parts in APropertyView. The reason is simple: You reuse APropertyView for the property creation and property update. Moreover, the type of Data Access Builder part used in each case is quite different:

- ❑ In the property creation process, the data flows from the view, where the user enters the property information, to the Data Access Builder parts (APropertyCreateView—see Figure 94 on page 262 and Figure 95 on page 264). The Data Access Builder parts are embedded in APropertyCreateView to represent the new records that are added to the database.
- ❑ In the property update process, the data flows from the Data Access Builder part variables to the view where the user can update the property information (APropertyUpdateView—see

Figure 101 on page 275). The Data Access Builder part variables are embedded in APropertyUpdateView to represent the records that already exist in the database.

Both APropertyCreateView and APropertyUpdateView reuse APropertyView to collect the user information. However, you cannot embed Data Access Builder parts or Data Access Builder part variables in APropertyView because:

- ❑ If you embed Data Access Builder parts, you could reuse APropertyView for APropertyCreateView but not for APropertyUpdateView.
- ❑ If you embed Data Access Builder part variables, you could reuse APropertyView for APropertyUpdateView but not for APropertyCreateView.

The above example reveals the kind of trade-off you must make if you want a part to be truly reusable.

## APropertyCreateView

From the CRC card, we know that APropertyCreateView part is responsible for creating a new property. From the design object model of the Property subsystem, we know that to create a property we must create each of its components: PropertyLog, AMarketingInfo, Multidoc, and Address. Each component has a corresponding class generated by Data Access Builder, namely, Property\_log, Marketing\_info, Multidoc, and Prop\_address (see Figure 93). These parts are the collaborators of APropertyCreateView.

The event-trace diagram for the creation of a property provides the information for connecting the different parts involved in the creation process. A first group of connections initializes each component (Address, Marketing\_info, Property\_log, Multidoc, and Property itself) as the user enters the property information from the view. These connections are attribute-to-attribute connections from the promoted attributes of APropertyView to the corresponding attributes in the Data Access Builder parts.

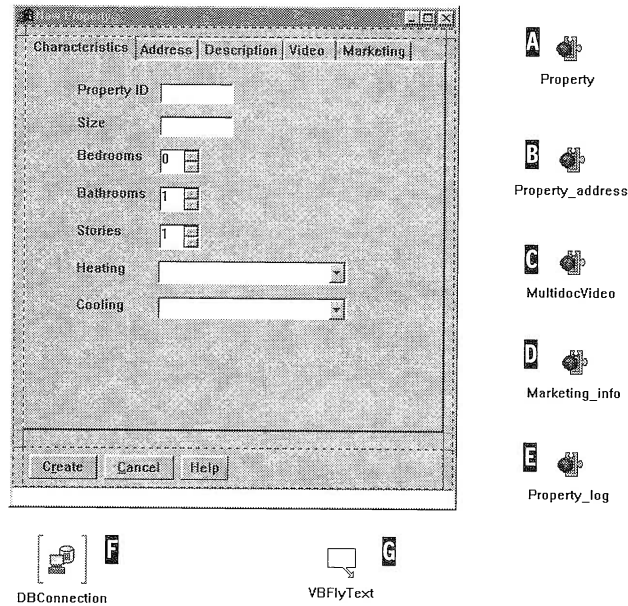
The second group of connections is triggered as soon as the user clicks on the **Create** push button and adds the contents of each Data Access Builder part to its respective table. These connections are event-to-action connections from the *Create* push button of APropertyCreateView to each Data Access Builder part.

In the sections that follow, you build the part in three steps:

1. You add the parts required to implement the part logic.






2. You build the attribute-to-attribute connections.
3. You build the event-to-action connections.



**Figure 93.** APropertyCreateView and Its Subparts

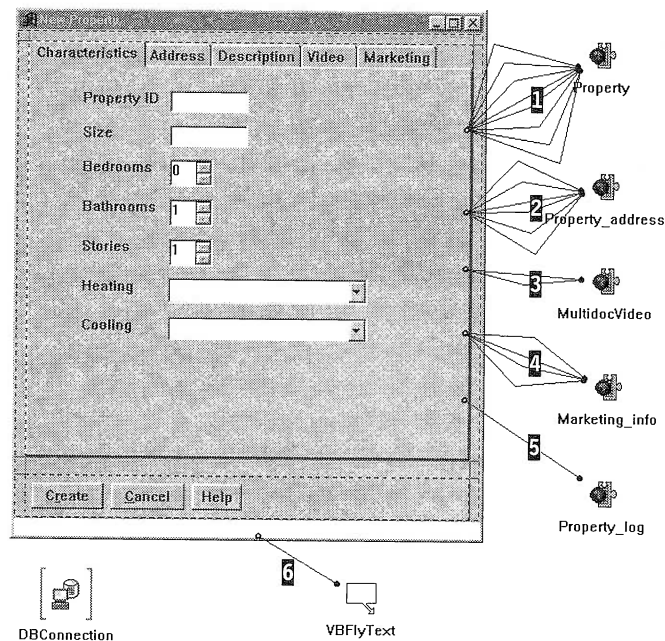
To add the parts required to implement the part logic, follow the step-by-step instructions in Table 36. Be sure your Data Access Builder parts are loaded in the Visual Builder Window.

Table 36. (Part 1 of 2) Adding Parts in APropertyCreateView	
Step	Action
1	Open the APropertyCreateView part.
2	Add a Property* part,  , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This part is initialized with the information on the Characteristics and Description pages.
3	Add a Property_address* part,  , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This part is initialized with the information on the Address page.
4	Add a Multidoc* part,  , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This part is initialized with the information on the Video page.

**Table 36.** (Part 2 of 2) Adding Parts in APropertyCreateView

Step	Action
5	Add a Marketing_info* part, <b>D</b> , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This part is initialized with the information on the Marketing page.
6	Add a Property_log* part, <b>E</b> , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This part is set up automatically with a custom logic connection.
7	Add an IDatastore* variable part, <b>F</b> , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This variable part holds the database connection. The connection is established when the application starts (see “Managing the Database Connection” on page 269 and “Using Variable Parts” on page 266).
<b>Note:</b> You can also add a variable part from the <i>Models</i> category. If you do, you must change the type of the variable to <b>IDatastore*</b> (option <b>Change Type...</b> in the variable part pop-up menu). Notice that the IVBVariablePart* part is called the IVBVariable* part in the parts palette. In the rest of this book, we call a variable from its reference name: IVBVariablePart.	
8	Promote the IDatastore* variable part so that it can get its contents from other parts (see “Managing the Database Connection” on page 269).
9	Add an IVBFlyText* part, <b>G</b> , to the free-form surface (located in the <i>Other</i> category). This part is used to display contextual help text in the window info area (see “Adding Fly-over Help to a Control” on page 270).
<b>Note:</b> Reverse highlighted letters are keyed to Figure 93 on page 260.	

Once you have placed the parts on the free-form surface, you can build the connections. First, build the group of attribute-to-attribute connections that associate the contents of each entry field of the view with the corresponding Data Access Builder part’s attribute (see Figure 94). Then, connect the IVBFlyText part to the window info area.



**Figure 94.** Attribute-to-Attribute Connections in APropertyCreateView

Make the attribute-to-attribute connections as shown in Table 37.

Table 37. (Part 1 of 2) Making Attribute-to-Attribute Connections in APropertyCreateView		
Key	Connection	Description
1	<ul style="list-style-type: none"><li><input type="checkbox"/> property_id → propertyID</li><li><input type="checkbox"/> propertySize → size</li><li><input type="checkbox"/> propertyBedrooms → bedrooms</li><li><input type="checkbox"/> propertyBathrooms → bathrooms</li><li><input type="checkbox"/> propertyStories → stories</li><li><input type="checkbox"/> propertyHeating → heating</li><li><input type="checkbox"/> propertyCooling → cooling</li><li><input type="checkbox"/> multiLineEditDescription-Text → description</li></ul>	Synchronize the value of each attribute (Property part).

**Table 37.** (Part 2 of 2) Making Attribute-to-Attribute Connections in APropertyCreateView

Key	Connection	Description
<b>2</b>	<input type="checkbox"/> address_id → propertyID <input type="checkbox"/> addressViewEntryFieldAreaText → area <input type="checkbox"/> addressViewEntryFieldCityText → city <input type="checkbox"/> addressViewEntryFieldStreetText → street <input type="checkbox"/> addressViewEntryFieldZipCodeText → zip_code <input type="checkbox"/> addressViewComboBoxStateText → state	Synchronize the value of each attribute (Property_address part). The address_id → propertyID connection enables the Property and Address tables to be joined.
<b>3</b>	<input type="checkbox"/> propertyID → multidoc_id <input type="checkbox"/> filename → VideoFileName	Synchronize the value of each attribute (MultidocVideo part). The propertyID → multidoc_id connection enables the Property and Multidoc tables to be joined.
<b>4</b>	<input type="checkbox"/> propertyID → property_id <input type="checkbox"/> commissionRate → commission_rate <input type="checkbox"/> downPaymentRate → down_payment_rate <input type="checkbox"/> price → price	Synchronize the value of each attribute (Marketing_info part). The propertyID → property_id connection enables the Property and Marketing_info tables to be joined.
<b>5</b>	propertyID → property_id	For joining the Property and Property_log tables.
<b>Note:</b> Fly-over help is detailed in “Adding Fly-over Help to a Control” on page 270.		
<b>6</b>	this → longTextControl	Display the long text of fly-over help in the window area.
<b>Note:</b> Reverse highlighted numbers are keyed to Figure 94 on page 262. To keep the drawings simple, we do not key all connections.		

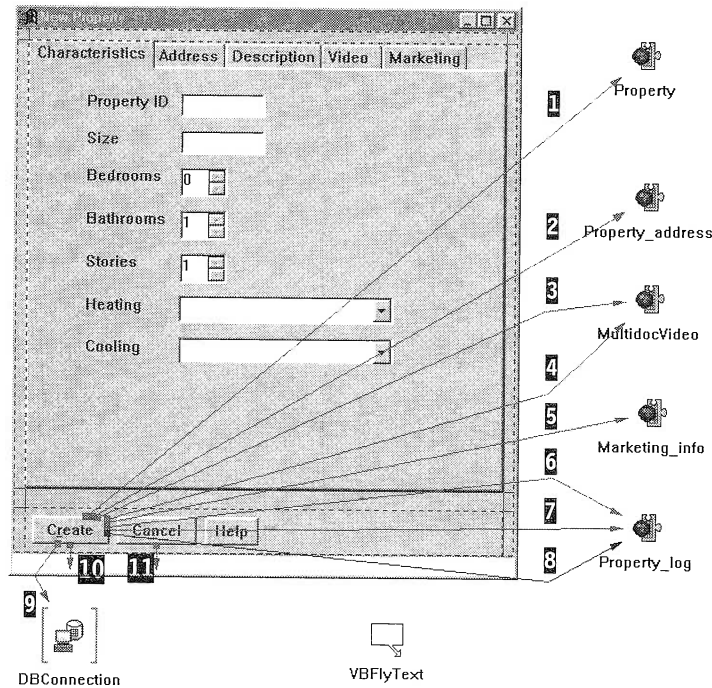
If you look closely at the event-trace diagram of the property creation scenario (Figure 37 on page 97), you can see that all of the attribute-to-attribute connections refer to the initialize arrows between the APropertyCreateView and each Data Access Builder class. Of course in the event-trace diagram, we do not represent all of the arrows. Rather, we draw one initialize arrow for several attribute-to-attribute connections

# Tip



To browse a set of connections all at once (either to see the connection order or to display the list of all connections without having to select connections one by one and look at the status line), select the reorder connection feature. From a selected part, click on the right mouse button and select the *Reorder Connections From* option from the part's pop-up menu. You will get a list of all connections issued from that part. You can change the order of the connections by dragging and dropping the connections within the list (see Figure 96 on page 266).

Now you can build the event-to-action connections, which refer to the add arrows drawn from APropertyCreateView to each Data Access Builder class (see Figure 95 and Table 38).

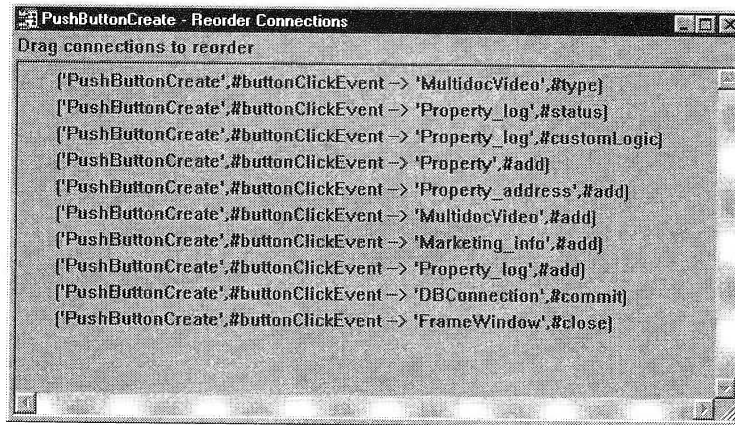


**Figure 95.** Event-to-Action Connections in APropertyCreateView

**Table 38.** Making Event-to-Action Connections in APropertyCreateView

Key	Connection	Description
<b>1</b>	buttonClickEvent → add	Add a row in the Property table.
<b>2</b>	buttonClickEvent → add	Add a row in the Property_address table.
<b>Note:</b> The order of the next two connections is crucial!		
<b>3</b>	buttonClickEvent → type	Set the document type. VIDEO is the only type implemented (see “Passing a Parameter to a Connection Statically” on page 271).
<b>4</b>	buttonClickEvent → add	Add a row in the Multidoc table.
<b>5</b>	buttonClickEvent → add	Add a row in the Marketing_info table.
<b>Note:</b> In the next three connections buttonClickEvent → add must be the last connection!		
<b>6</b>	buttonClickEvent → status	Set the property status to AVAILABLE at creation time (see “Passing a Parameter to a Connection Statically” on page 271).
<b>7</b>	buttonClickEvent → custom-Logic	Set the time stamp creation (see “Using Custom Logic” on page 271).
<b>8</b>	buttonClickEvent → add	Add a row in the Property_log table.
<b>9</b>	buttonClickEvent → commit	Commit the transaction in the database.
<b>10</b>	buttonClickEvent → close	Close the window.
<b>11</b>	buttonClickEvent → close	Close the window.
<b>Note:</b> Reverse highlighted numbers are keyed to Figure 95 on page 264.		

Ensure that the event-to-action connections from the *Create* push button are in the proper order. To check the order of connections drawn from a part, use the **Reorder Connections From...** option from the part's pop-up menu (see Figure 96).



**Figure 96.** Connection Order for the Create Push Button

You can now save your part and generate its code. From the Visual Builder window, select **Save and generate** → **Part source** in the **File** pull-down menu.

## Using Variable Parts

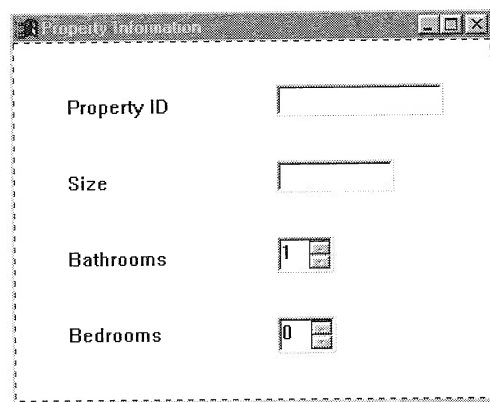
Variable parts play an important role in applications. You use variable parts primarily in two situations:

- ❑ To act as a placeholder inside a composite part for parts that cannot be found in the composite part. Using variable parts in nonvisual parts enables you to pass data or functions between parts. Using variable parts in visual parts enables you to pass data across different visual parts. For example, you use an *IDatastore\** variable part to transmit the connection from one visual part of the application to another (see “Managing the Database Connection” on page 269). You will also use variable parts in the *APropertySearchResultView* and *APropertyUpdateView* to pass the record selected in the first view to the second view.
- ❑ To represent part instances created with factory parts (see “Using an Object Factory to Update the Database” on page 295).

You can add a variable part on the free-form surface in three different ways:

- ❑ Select a variable part from the *Models* category. Once the variable part is on the free-form surface, you must change its type to the type of the part it represents.
- ❑ From the *Composition Editor* menu, use **Options** → **Add part...** to add a part. Instead of selecting *Part* in the *Add as* radio button, you select *Variable*. In this case the type of the variable part is automatically set according to the part class you have entered.
- ❑ Use the tear-off feature to expose a subpart of a part (see “Tearing Off an Attribute” on page 344). In this case, the type of the variable part is automatically set according to the class of the attribute exposed.

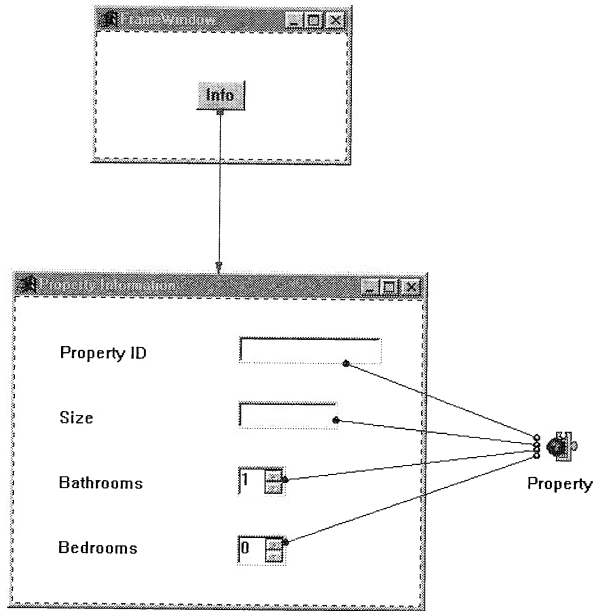
It is often better to use a variable part to update a set of entry fields than it is to promote *text* attributes of entry fields and then update the entry field contents one by one from another part by using attribute-to-attribute connections. For example, suppose you build a simple property view to display some property information (Figure 97). You would use the Data Access Builder Property part to hold the property information.



**Figure 97.** Simple View to Display Property Information

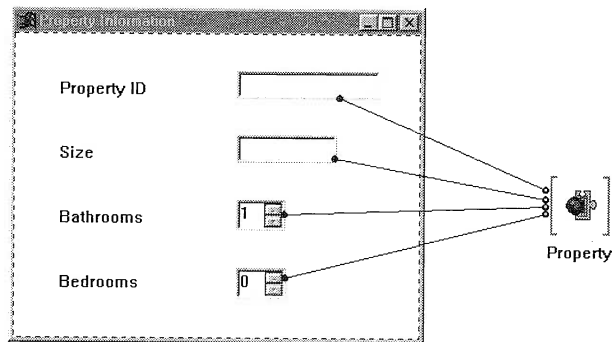
Each time you want to reuse the Property view from another part and update the contents of each entry field with the property information, you must connect each attribute from the Property part to the *text* attribute of each entry field. In this case, you must promote the *text* attribute of each entry field of the Property view (Figure 98).





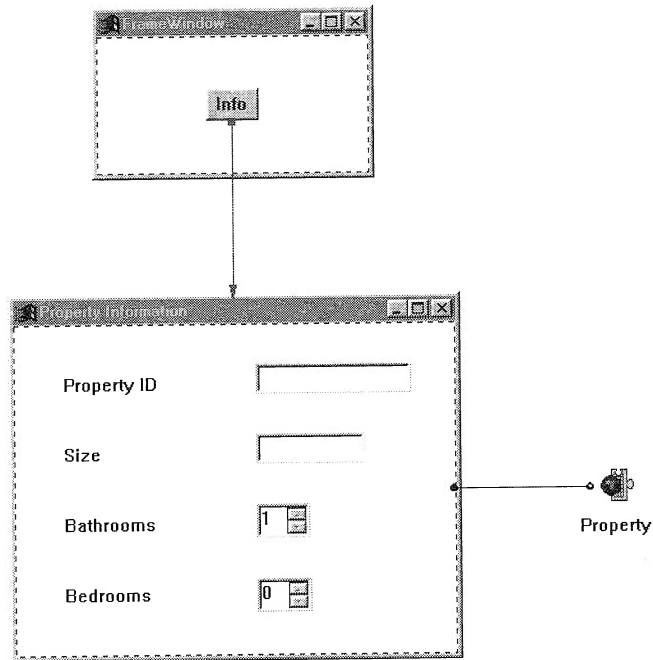
**Figure 98.** Reusing the Property View from Another Part: First Try

A better design would be to use a Property variable part in the Property view and promote only the variable part itself (Figure 99).



**Figure 99.** Simple Property View with Its Associated Variables

In this way, when you use the Property view from another part, you draw only one connection to update the view (Figure 100).



**Figure 100.** Reusing the Property View from Another Part: Second Try

In our application, you use variable parts in `APropertyUpdateView` to set the contents of entry fields in each notebook page. You set the variable contents from `APropertySearchResultView`, using only one connection (see “`APropertySearchResultView`” on page 289).

## Managing the Database Connection

Before the user can access a subsystem, a database connection must be established (see Chapter 6, “Mapping Relational Tables Using Data Access Builder,” on page 131). A problem arises when the application accesses the database from several views. In that case, each view must hold an `IDatastore*` part to connect to the database, and the application ends up with some overhead each time the user switches from one view to another, chaining database connections all along. To avoid this problem, use only one `IDatastore*` part in your application primary view and pass it to the other views through variable parts. This is a convenient and efficient way of managing a unique connection shared among several application views.

Each part using the database must contain a variable part of type `IDatastore*`. This variable part must be promoted to receive an `IDatastore*` reference from the parts that use it.

Now, let us suppose that your application consists of several parts that are organized in a five-level hierarchy. If some parts of the last level must access the database, you must add and promote an `IDatastore*` variable part in all intermediate levels from the second level up to the fifth. This action is necessary because the promote feature cannot jump a hierarchy level.

In our case, you add an `IDatastore*` part in the primary view, `ARealMainView`, to establish one database connection at startup. Then, because this database connection must be used in `APropertyUpdateView`, you must add and promote an `IDatastore*` variable part in each view that belongs to the hierarchy branch: `APropertySearchParameterView`, `APropertySearchResultView`, and `APropertyUpdateView` (see the view hierarchy in Figure 50 on page 149).

## Adding Fly-over Help to a Control

To provide instant help to the novice user, you can use *fly-over help*, which consists of a text string that is displayed when the user positions the mouse pointer over a control, such as an entry field or a push button. The text should be short and should indicate the purpose of the control.

The application can provide two kinds of fly-over help

- ☐ A short text string that your application displays in a bubble (it is also called *bubble help*) next to the subpart on which you have your mouse pointer
- ☐ A long text string (more explanatory) that your application displays in a text control (such as an entry field or an info area)

To provide fly-over help for a subpart:

1. Drop an `IVBFlyText*` part on the free-form surface (`IVBFlyText*` part is located in the *Other* category).
2. Open the settings of the subpart and, on the control page, enter the fly-over short text, fly-over long text, or both.

Text entered in fly-over short text is displayed as a bubble help for this subpart, and you do not need any connections to make it work!

Text entered in fly-over long text can be displayed in a window info area: Connect the **this** attribute of the info area to the longTextControl of the IVBFlyText\* part (for more information refer to the *Visual Builder User's Guide*).

To add fly-over help to the *Create* push button, follow these steps:

1. Open the settings notebook of the *Create* push button.
2. Switch to the Control page.
3. In the *Fly over short text* entry field, type **Create a new property**. This text will display in a bubble when the user positions the mouse on the *Create* push button.
4. In the *Fly over long text* entry field, type **Fill in each entry field before creating a property in the database**. This text will display in the frame window info area added for this purpose.

These four steps are all you need to give your application slick contextual help! Save the part again and regenerate its source code to register the changes.

In the next sections, feel free to add your own fly-over short and long texts for your parts.

## Passing a Parameter to a Connection Statically

As mentioned in “Passing a Parameter to a Connection Dynamically” on page 250, you can provide a connection with parameters either dynamically or statically. In connection 3 (see Table 38 on page 265), you provide statically the string, VIDEO, as the type of medium to which the file name refers. Double-click with the mouse on this connection to get the connection settings dialog box. Then click on the **Set Parameters...** push button. You can see VIDEO in the type entry field (notice that there are no quotes). Do the same for the property status, setting the **status** attribute of the Property\_log\* part to AVAILABLE (other property statuses are PENDING and SOLD).

## Using Custom Logic

A custom logic connection is an easy way of calling your own customized C or C++ code whenever an attribute's value changes or an event occurs. When you associate an attribute with a custom logic connection, the attribute event identifier is used to notify the custom logic connection and call your code when the attribute's value changes. You can use this connection for small pieces of code that you do not plan to reuse. For more information about custom logic connections, refer to the *Visual Builder User's Guide*.

In APropertyCreateView, you use a custom logic connection to set the *download\_timestamp* and *last\_update* attributes of the Property\_log table.

A *download\_timestamp* is associated with the creation time and date of each property record in the database. A *last\_update* attribute is associated with the date and time of the update of each property record in the database. It contains the date and time when the record is created or updated in the database. In DB2 for Windows, the *creation-time-stamp* attribute is a string of 26 characters with the following format:

yyyy-mm-dd-hh.mm.ss.xxxx

where

- ☐ yyyy represents the year (for example, 1995)
- ☐ mm represents the month (for example, 06 for June)
- ☐ dd represents the day (for example, 23 for the twenty-third)
- ☐ hh.mm.ss.xxxx represents the time (format: hour.minute.second.millisecond)

IDate and ITime parts are combined to make up the time stamp as follows:

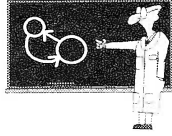
**Code to Store Current Time Stamp**

```
target → setDownload_timestamp(  
    IString(IDate::today().asString("%Y-%m-%d"))+"-"+  
    IString(ITime::now().asString("%H.%M.%S")));  
target → setLast_Update(  
    IString(IDate::today().asString("%Y-%m-%d"))+"-"+  
    IString(ITime::now().asString("%H.%M.%S")));
```

IDate::today() returns the current date. The asString() method is used to translate the current date in the yyyy-mm-dd format that is compatible with the DATE type of DB2 tables (for more information, see the *IBM Open Class Library Reference*).

ITime::now() returns the current time, and the asString() method is used to translate it into the correct DB2 time format, hh-mm-ss. The same code is applied to the *last\_update* attribute because at creation time, the creation\_time stamp and last\_update time stamp are the same.

## Technical Information



The `IDate` and `ITime` parts are class interface parts. As mentioned in “AMarketingInfo” on page 226, class interface parts are nonvisual parts that have no notification ability. In other words, they cannot send events to other parts. You can trigger their action from your application events, however. Visual Builder provides many class interface parts that conveniently wrap standard classes from the IBM Open Class Library to enable you to use the classes in the Composition Editor. In the same way, you can use your own C++ classes in Visual Builder as class interface parts (refer to Chapter 8, “Creating Nonvisual Parts,” on page 225 and Chapter 10, “More about Visual Builder...,” on page 353).

Notice that you must use the `setDownload_timestamp` and `setLast_Update` methods to change the attribute value of `Property_log`. Otherwise, the target part will never be notified that the attribute has changed.

### Tip



The target of the custom logic connection is the `Property_log*` part. Therefore you call its methods by using the `target → function(param1,...)` expression. Sometimes your custom logic code may require that you use more than one part. To access the parts, you must select the free-form surface as the target connection. The free-form surface represents the composite part itself, and each of its subparts is known as a pointer within the part. In the code generated by Visual Builder, the identifier of a pointer to a subpart is the name of the subpart with the letter `i` as a prefix. For example, in `APropertyCreateView`, if you had used the free-form surface (which represents `APropertyCreateView*` itself) as the target for the custom logic connection, you would have written the snippet code as follows:

```
target → iProperty_log → setDownload_timestamp(
    IString(IDate::today().asString("%Y-%m-%d"))+"-"+
    IString(ITime::now().asString("%H.%M.%S")));
target → iProperty_log → setLast_Update(
    IString(IDate::today().asString("%Y-%m-%d"))+"-"+
    IString(ITime::now().asString("%H.%M.%S")));
```

Because you use the `IDate` and `ITime` parts in your visual part, you must include their corresponding header files in the code generated by Visual Builder:

1. Switch to the Class Editor of APropertyCreateView.
2. In the *User files included in generation* group box fill in the *Required include files* list box as follows:

```
<idate.hpp> _IDATE_  
<itime.hpp> _ITIME_
```

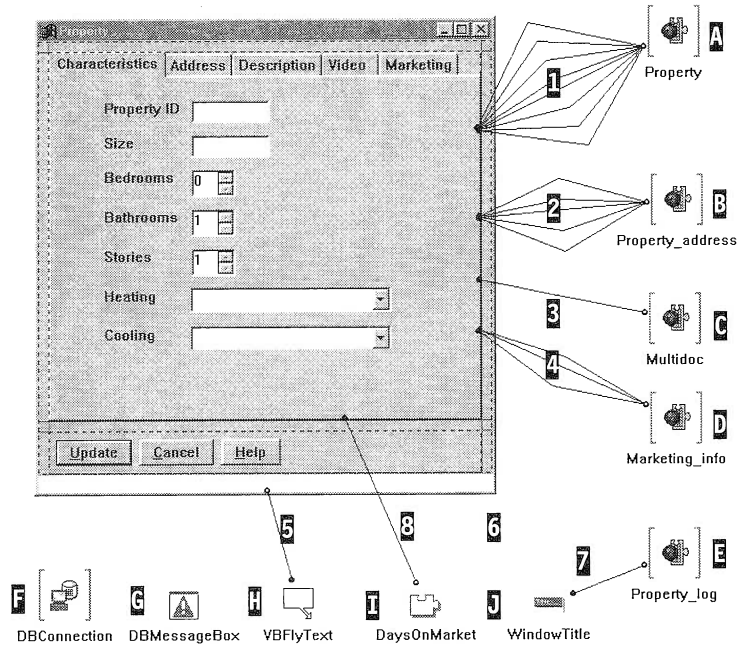
The idate.hpp and itime.hpp header files will be included in the APropertyCreateView header file. The preprocessor uses the `_IDATE_` and `_ITIME_` variables to avoid including the headers if they already have been included by other subparts of the view.

You can now save your part and generate its code. From the Visual Builder window, select **Save and generate** → **Part source** in the *File* pull-down menu.

## APropertyUpdateView

APropertyUpdateView is used to update the property information. In many ways, APropertyUpdateView is similar to APropertyCreateView. However, unlike APropertyCreateView, the property records of APropertyUpdateView are not created across the five relational tables. They are updated across these five tables.

An instance of APropertyUpdateView is created when the user has already selected a property from the container. (This container displays the result of a search in APropertySearchResultView.) The significant difference in the update process is that the property and its components already exist. For this reason, when you add the Data Access Builder parts on the free-form surface, you must add them as variable parts, not as parts (Figure 101). The contents of the variable are set by parameter connections in APropertySearchResultView (see “Using Variable Parts” on page 266).



**Figure 101.** Attribute-to-Attribute Connections in APropertyUpdateView.






To add the different subparts to the free-form surface, follow the steps in Table 39. (Letters refer to Figure 101.)

**Table 39.** (Part 1 of 3) Adding Subparts in APropertyUpdateView

Step	Action
1	Open the APropertyUpdateView part.
2	Add a Property* variable part, <b>A</b> , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This variable receives its contents from APropertySearchResultView and sets the Characteristics and Description page information.
3	Add a Property_address* variable part, <b>B</b> , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This variable gets its contents from APropertySearchResultView and sets the Address page information.
4	Add a Multidoc* variable part, <b>C</b> , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This variable gets its contents from APropertySearchResultView and sets the Video page information.



**Table 39.** (Part 2 of 3) Adding Subparts in APropertyUpdateView

Step	Action
5	Add a Marketing_info* variable part,  , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This variable gets its contents from APropertySearchResultView and sets the Marketing page information.
6	Add a Property_log* variable part,  , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This part is automatically updated with a custom logic connection.
7	Add an IDatastore* variable part,  , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This variable holds the database connection that established when the application starts (see “Using Variable Parts” on page 266 and “Managing the Database Connection” on page 269).
<b>Note:</b> You can also add a variable part from the <i>Models</i> category. If you do so, you must change the type of the variable to the type of the part it represents (use the <b>Change Type...</b> option in the variable part pop-up menu).	
9	Promote the Property* variable part.
10	Promote the Property_address* variable part.
11	Promote the Multidoc* variable part.
12	Promote the Marketing_info* variable part.
13	Promote the Property_log* variable part.
14	Promote the IDatastore* variable part.
<b>Note:</b> Each variable part is promoted to be accessible from other parts and to receive its contents from these parts (see “Using Variable Parts” on page 266).	
15	Add an IMessageBox* part,  , on the free-form surface (located in the <i>Other</i> category). This message box is used to display an exception if the transaction cannot be committed (see “Showing Exception in a Message Box” on page 282).
16	Add an IVBFlyText* part,  , on the free-form surface (located in the <i>Other</i> category) and add some fly-over short and long texts to the different push buttons (see “Adding Fly-over Help to a Control” on page 270).

**Table 39.** (Part 3 of 3) Adding Subparts in APropertyUpdateView

Step	Action
17	Add an IVBLongPart* part, <b>1</b> , to the free-form surface ( <b>Option</b> → <b>Add Part...</b> from the Composition Editor). This part contains the number of days since the property was created in the database (see “Using a Member Function Connection” on page 283).
18	Add an ITitle* part, <b>2</b> , to the free-form surface (located in the <i>Frame Extensions</i> category). This part represents the frame window title. Its <i>objectText</i> attribute is updated by the <i>last_update</i> attribute of Property_log (see “Updating a Window Title Dynamically” on page 286).
<b>Note:</b> Reverse highlighted letters are keyed to Figure 101 on page 275.	

Because APropertyUpdateView is similar to APropertyCreateView, we use the same mode of presentation: First we describe the attribute-to-attribute connections, then we describe the event-to-action, parameter, and custom logic connections.

Connect the parts with the attribute-to-attribute connections shown in Table 40.

**Table 40.** (Part 1 of 2) Making Attribute-to-Attribute Connections in APropertyUpdateView

Key	Connection	Description
<b>1</b>	<input type="checkbox"/> propertyID → property_id <input type="checkbox"/> size → propertySize <input type="checkbox"/> bedrooms → propertyBedrooms <input type="checkbox"/> bathrooms → propertyBathrooms <input type="checkbox"/> stories → propertyStories <input type="checkbox"/> heating → propertyHeating <input type="checkbox"/> cooling → propertyCooling <input type="checkbox"/> description → multiLineEditDescriptionText	Synchronize the value of each attribute of Property part with the Characteristics page entry fields.

**Table 40.** (Part 2 of 2) Making Attribute-to-Attribute Connections in APropertyUpdateView

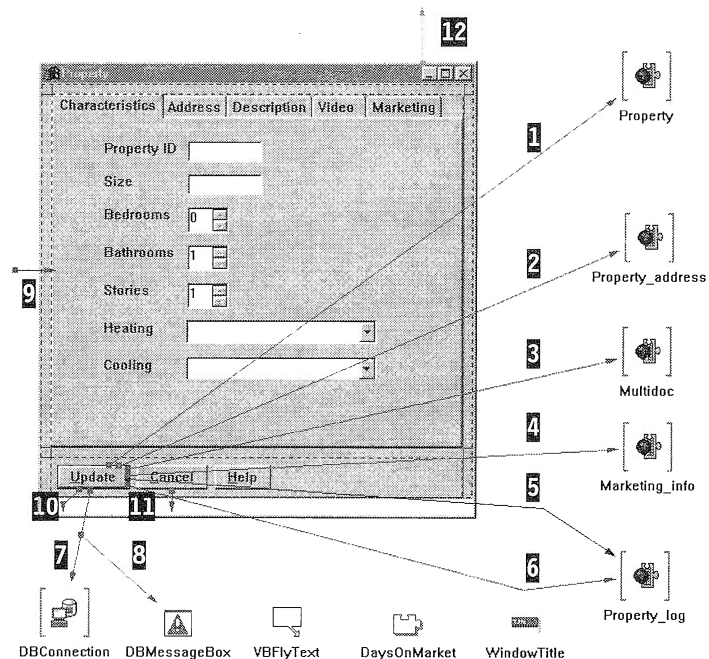
Key	Connection	Description
<b>2</b>	<input type="checkbox"/> area → addressViewEntryFieldAreaText <input type="checkbox"/> city → addressViewEntryFieldCityText <input type="checkbox"/> street → addressViewEntryFieldStreetText <input type="checkbox"/> zip_code → addressViewEntryFieldZipCodeText <input type="checkbox"/> state → addressViewComboBoxStateText	Synchronize the value of each attribute of Property_address part with the Address page entry fields. The propertyID → property_id connection enables the Property and Address tables to be joined.
<b>3</b>	<input type="checkbox"/> VideoFileName → filename	Synchronize the value of each attribute of MultidocVideo part with the Video page entry fields. The propertyID → property_id connection enables the Property and Multidoc tables to be joined.
<b>4</b>	<input type="checkbox"/> commission_rate → commissionRate <input type="checkbox"/> down_payment_rate → downPaymentRate <input type="checkbox"/> price → price	Synchronize the value of each attribute of Marketing_info part with the Video page entry fields. The propertyID → property_id connection enables the Property and Marketing_info to be joined.
<b>5</b>	this → longTextControl	Display the long text of fly-over help in the window area.
<b>6</b>	this → owner	Associate the window title control with the main window.
<b>7</b>	last_update → viewText	Synchronize the value of last_update with the contents of the window title.
<b>8</b>	valueAsText → DaysOnMarket	Synchronize the value of the DaysOnMarket part with the DaysOnMarket feature promoted from PropertyView.

**Note:** Reverse highlighted numbers are keyed to Figure 101 on page 275. To keep the drawings simple, we do not key all of the connections.

Because the event-trace diagram of the property update scenario is not detailed enough, you cannot see the attribute-to-attribute connections (Figure 39 on page 99). Rather, you see a connection from APropertySearchResultView and APropertyUpdateView with the *send info* label. This connection is the representation of the data which flows from one view to another by means of the variable parts (see “Using Variable Parts” on page 266).

Notice that the Property identifier is propagated into the EntryField-PropertyID of APropertyView by using the *propertyID* → *property\_id* attribute-to-attribute connection (see 1 in Table 40). However, to prevent the user from modifying the property identifier, EntryField-PropertyID is set to *read only* by triggering its *disableDataUpdate* (see connection 12 in Figure 102 and Table 41). For this reason, there is no other attribute-to-attribute connection from the value of this entry field to the other Data Access Builder parts (see 2, 3, and 4).

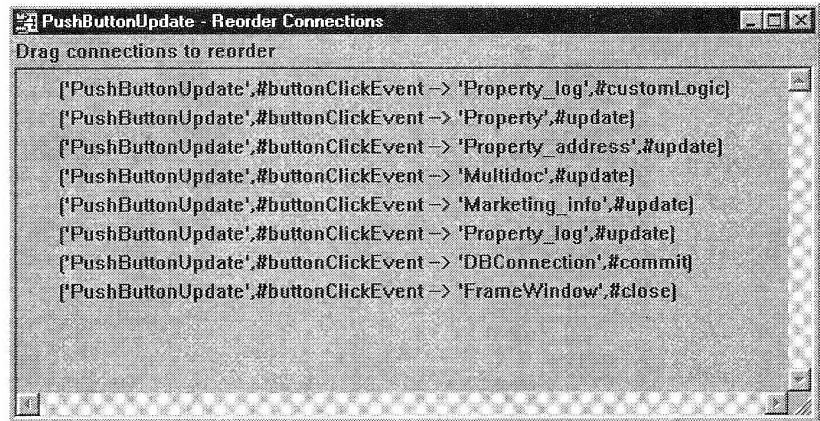
Using event-to-action, parameter, custom logic and event-to-member function connections, connect the subparts as shown in Figure 102 and Table 41.



**Figure 102.** Event-to-Action Connections in APropertyUpdateView

<b>Table 41.</b> Making Event-to-Action Connections in APropertyUpdateView		
<b>Key</b>	<b>Connection</b>	<b>Description</b>
<b>1</b>	buttonClickEvent → update	Update the row in the Property table.
<b>2</b>	buttonClickEvent → update	Update the row in the Property_address table.
<b>3</b>	buttonClickEvent → update	Update the row in the Multidoc table. Notice that you do not have to set the type to VIDEO because it has already been set, and it cannot be changed from the view.
<b>4</b>	buttonClickEvent → update	Update the row in the Marketing_info table.
<b>Note:</b> For the next two connections, buttonClickEvent → update must be the last connection!		
<b>5</b>	buttonClickEvent → custom-Logic	Set the last update time stamp (see “Using Custom Logic” on page 271).
<b>6</b>	buttonClickEvent → update	Update the row in the Property_log table.
<b>7</b>	buttonClickEvent → commit	Commit the transaction in the database.
<b>8</b>	exceptionOccurred → showException	Show the commit exception in the message box when it occurs.
<b>9</b>	visible → daysOnMarket()	Compute the number of days the property is on the market when the window is displayed (see “Using a Member Function Connection” on page 283).
<b>10</b>	buttonClickEvent → close	Close the window.
<b>11</b>	buttonClickEvent → close	Close the window.
<b>12</b>	ready → propertyIDReadOnly	Set the property identifier entry field for read only.
<b>Note:</b> Reverse highlighted numbers are keyed to Figure 102 on page 279.		

Ensure that the event-to-action connections from the *Update* push button are in the proper order. To check the order of connections drawn from a part, use the **Reorder Connections From...** option from the part's pop-up menu and reorganize the connections displayed in the window (see Figure 103).



**Figure 103.** Connection Order for the Update Push Button

Because *IDate*, *ITime*, *IString*, and the *IString* parser parts are used through connection **10** to calculate the number of days the property is on the market, you must include their corresponding header files in the code generated by Visual Builder:

1. Switch to the Class Editor of *APropertyUpdateView*.
2. In the group box entitled *User files included in generation*, fill in the *Required include files* list box as follows:

```
<idate.hpp>      _IDATE_
<itime.hpp>      _ITIME_
<istring.hpp>    _ISTRING_
<istparse.hpp>  _ISTPARSE_
```

These header files are included in the *APropertyUpdateView* header file. The processor uses `_IDATE_`, `_ITIME_`, `_ISTRING_`, and `_ISTPARSE_` variables to avoid including these headers if they are already included by other subparts of the view.

You can now save your part and generate its code. From the Visual Builder window, select **Save and generate** → **Part source** in the *File* pull-down menu.

As with APropertyCreateView, you can see the correspondence of the event-to-action *update* in the view and its representation in the event-trace diagram of the property update scenario (Figure 39 on page 99).

## Showing Exception in a Message Box

To trigger the display of a message box when a part throws an exception,

1. Add an IMessageBox\* part to the free-form surface.
2. Open the message box settings and set the title to your liking.
3. Connect the **exceptionOccurred** event of the event-to-action connection that throws the exception to the **showException** action of the message box.

When the connection is triggered, the action is executed in a *try* block. If the action fails, a *catch* block allows the connection to execute an alternative action, such as showing the message box (for more information on the C++ exception-handling framework, refer to the *C/C++ Language Reference*). This facility is used in APropertyUpdateView to catch a commit exception on the database. In the same way, the application can display a message box when an exception occurs during the update of the database. In this case, a rollback of the database can be initiated by connecting the **exceptionOccurred** event of the connection that throws the exception to the **rollback** action of the IDatastore part.

You can also tailor the message box to your needs, using the **show** action. Open the settings of the *exceptionOccurred* → *show* connection and set the message and the severity of the exception (see “Using a Message Box to Display the Clause” on page 318).

### Tip



You can use the message box to trace the flow of your program and display some variable contents or some attribute values during the execution of the program. For example, suppose you want to trigger an **A** action when a push button is clicked. To check the contents of the variable on which the action depends, you connect the **buttonClickEvent** event of the push button to the **show** action of a message box and send the contents of the variable as a parameter to this connection. This event-to-action connection must be triggered before action **A** is executed (see “Using a Message Box to Display the Clause” on page 318 and Figure 123 on page 319).

## Using a Member Function Connection

The number of days the property is on the market is stored using IVBLongPart. This part is available from the VB SAMPLE.VBB file and contains many methods that you might find useful for your future applications (see also “Using Sample Parts” on page 248 for more information on the sample parts). The value hold by IVBLongPart is calculated by an *event-to-member function* connection. This connection can be a valuable alternative to a custom logic connection when you have to reuse your own C++ code within the same part or when the code is more than a few lines. There are two types of member function connections: the *event-to-member function* connection and the *attribute-to-member function* connection. Both types enable you to call a member function of your part that you declare and define in the specific hpv and cpv user files.

The member function connection may reveal some advantages over the custom logic connection:

- ❑ The code is written in cpv and hpv files, which are separated from the actual cpp and hpp files part that generates Visual Builder for the part. Thus, each time the visual part code is generated, it does not impact your member function code.
- ❑ In addition to a *public* access specifier, you can choose to declare the member functions with *private* or *protected* access specifiers. You may find this facility useful in the context of derivation in controlling the access of such “implementation functions” to inherited members.

In our example, you add a public method, void DaysOnMarket() (Figure 104), to APropertyUpdateView to calculate the number of days a property is on the market. This number is calculated from the current date and the creation time stamp of the property.

```

/*****
/*
/*  Declaration of the daysOnMarket member function
/*
/*  Description:
/*      This method calculates the time difference between
/*      the system date and the property creation date.
/*
/*****

public:
    void APropertyUpdateView::daysOnMarket();

```

**Figure 104.** DaysOnMarket Public Method Declaration



The public method converts the database time stamp to an `IDate` format and calculates the difference from the current date by using the `dateToday` function, which returns the system date (see Figure 105). Then, the method sets the contents of the *DaysOnMarket* part.

```

IDate dateToday;
IString day, month, year, temp;
iProperty_log
    ->target()
    ->download_timestamp() >> year
                        >> '_,'
                        >> month
                        >> '_,'
                        >> day
                        >> '_,'
                        >> temp;

IDate::Month monthYear;
switch (month.asInt()) {
    case 1:
        monthYear = IDate::January;
        break;
    case 2:
        monthYear = IDate::February;
        break;
    case 3:
        monthYear = IDate::March;
        break;
    case 4:
        monthYear = IDate::April;
        break;
    case 5:
        monthYear = IDate::May;
        break;
    case 6:
        monthYear = IDate::June;
        break;
    case 7:
        monthYear = IDate::July;
        break;
    case 8:
        monthYear = IDate::August;
        break;
    case 9:
        monthYear = IDate::September;
        break;
    case 10:
        monthYear = IDate::October;
        break;
    case 11:
        monthYear = IDate::November;
        break;
    case 12:
        monthYear = IDate::December;
        break;
} /* end switch */

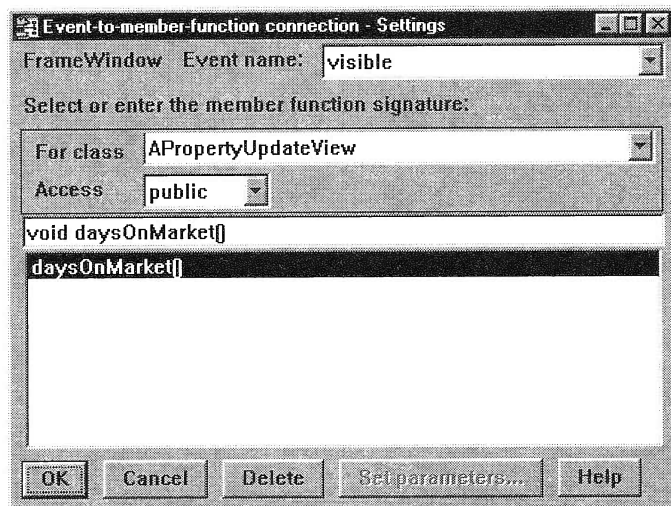
IDate creationDate = IDate(monthYear, day.asInt(), year.asInt());
iDaysOnMarket->setValue(dateToday - creationDate);

```

**Figure 105.** DaysOnMarket Public Method Definition

To build the event-to-member function action (see 9 in Table 41 on page 280), follow these instructions:

1. Select the *Connect* option from the IFrameWindow\* part pop-up menu and select the **visible** event. The pointer becomes a spider.
2. Move the pointer to the free-form surface and click with the left mouse button. Then choose the **More...** option from the pop-up menu. In the dialog box (Figure 106), you are prompted to select the access specifier and enter the member function prototype.
3. In the *Access* drop-down list, select **public** as the access specifier.
4. In the line below, enter the member function prototype: `void DaysOnMarket()`.
5. Click on the **OK** push button. A light green connection is drawn from the IFrameWindow\* part to the free-form surface.
6. Switch to the Class Interface Editor. In the *User files included in generation* group box, fill in the *hvp* and *cpv* entry fields with the respective values, `vrpupdv.hvp` and `vrpupdv.cpv`.
7. With your favorite editor, edit the *hvp* and *cpv* files as shown in Figure 104 on page 283 and Figure 105 on page 284.



**Figure 106.** Member Function Dialog Box

In Chapter 10, “More about Visual Builder...,” on page 353, we show you how to use the browser provided with VisualAge for C++ to avoid entering the prototype of your methods in the member function dialog box.

## Updating a Window Title Dynamically

The frame window extension category of the parts palette enables you to tailor your frame window. You have been shown how to add an info area or a menu to your frame window. Using an *ITitle\** part, you can change the title of the window after the window is created:

1. Add an *ITitle\** part to the free-form surface.
2. Specify the frame window to which the *ITitle\** part is related. You can consider two options:
  - Draw an attribute-to-attribute connection from the *this* attribute of the frame window to the *owner* attribute of the *ITitle\** part.
  - Open the *ITitle\** part settings and set the *owner* attribute to the name of the related window (do not forget to prefix the name with *i*; see “Using Custom Logic” on page 271).
3. Use the *ITitle\** part as follows:
  - If you want to change the title when an event occurs, connect the event to the **setTitleText** action of the *ITitle\** part and set the parameters required by the connection. (Click on the **Set Parameters...** push button in the connection settings and enter the title in the *objectName* parameter. You can optionally specify the view name and the view number.)
  - If you want to set the title from one or more attributes of other parts, connect a *text* attribute to the *objectText* or *viewText* attributes of the *ITitle\** part. You can optionally connect a numeric attribute to the *viewNumber* attribute of the *ITitle\** part.

In our case, you display the time stamp of the property creation by using an attribute-to-attribute connection from the *last\_update* attribute of *Property\_log* to the *viewText* attribute of the *Title* part (see connection 8 in Figure 101 on page 275). The *objectText* attribute of the *IFrameWindow\** part is directly set in the Composition Editor to “Property.”

## ADeleteDialogView

*ADeleteDialogView* visual part is used by each subsystem to warn users before they delete a record in the database. This view is so simple that step-by-step instructions are not needed.

There is only one connection to draw to complete the view:

1. Open `ADeleteDialogView`.
2. Connect the **buttonClickEvent** event of the *Cancel* push button to the **close** action of the frame window.

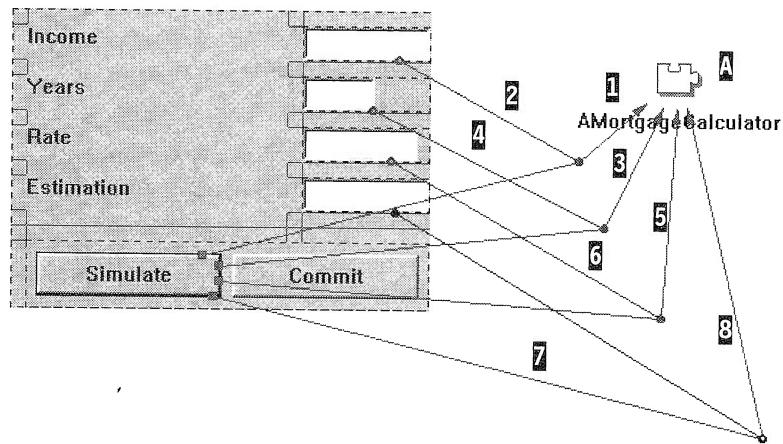
You can now save your part and generate its code. From the Visual Builder window, select **Save and generate** → **Part source** in the *File* pull-down menu.

Even this simple view gives us a good opportunity to point out that you should always close your frame window part by using an event-to-action connection within the part. This approach makes your part easier to reuse because it behaves independently of other parts. However, sometimes you need to draw the part's connections within a composite part that reuses your part. This is the case for the event-to action connection: **buttonClickEvent** event of the **OK** push button to the **close** action of the frame window. Indeed, when the user clicks the **OK** push button, an event-to-action connection must delete the records in the database before closing the window. Once the records are deleted, the **close** action must be triggered from an event-to-action connection drawn in the part that manages the record deletion. These two event-to-action connections are executed sequentially. Therefore, you must promote the **buttonClickEvent** event of the **OK** push button to access it from another part.

Of course, you could decide to include the parts involved in the deletion process of a property in `ADeleteDialogView`—such as the Data Access Builder parts—but, in this case, `ADeleteDialogView` would not have been reusable by the other subsystems. In effect, the parts involved in the deletion process vary from one subsystem to another.

## ASimulMortgageView

You build `ASimulMortgageView` by connecting to it `AMortgageCalculator` that you developed in Chapter 8, “Creating Nonvisual Parts,” on page 225. `AMortgageCalculator` is a class interface; by implementation, it is not notification enabled. For this reason you cannot use attribute-to-attribute connection with `ASimulMortgageView`. Instead, you use event-to-action and parameter connections as described in Table 42.



**Figure 107.** Connecting AMortgageCalculator with ASimulMortgageView

To connect AMortgageCalculator to ASimulMortgageView follow the steps in Table 42:

<b>Table 42.</b> (Part 1 of 2) Connecting AMortgageCalculator with ASimulMortgageView	
Step	Action
1	Open the ASimulMortgageView part.
2	Add an AMortgageCalculator* part, to the free-form surface <b>A</b> . (Use the <b>Option</b> → <b>Add part...</b> from the Composition Editor pull-down menu to add it to the free-form surface.)
3	Connect the <b>buttonClickEvent</b> of the PBSimulate push-button to the <b>income</b> action of AMortgageCalculator, <b>1</b> .
4	Connect the <b>valueAsDouble</b> attribute of the income entry field to the <b>aIncome</b> parameter of the connection <b>1</b> , (see connection <b>2</b> ).
5	Connect the <b>buttonClickEvent</b> of the PBSimulate push-button to the <b>years</b> action of AMortgageCalculator, <b>3</b> .
6	Connect the <b>valueAsDouble</b> attribute of the years entry field to the <b>aYears</b> parameter of the connection <b>3</b> (see connection <b>4</b> ).
7	Connect the <b>buttonClickEvent</b> of the PBSimulate push-button to the <b>rate</b> action of AMortgageCalculator, <b>5</b> .

**Table 42.** (Part 2 of 2) Connecting AMortgageCalculator with ASimulMortgageView

Step	Action
8	Connect the <i>valueAsDouble</i> attribute of the rate entry field to the <i>aRate</i> parameter of the connection 5, (see connection 5).
9	Connect the <i>buttonClickEvent</i> of the PBSimulate push-button to the <i>estimate</i> action of AMortgageCalculator, 7.
10	Connect the <i>actionResult</i> attribute from the connection 7 to the <i>valueAsDouble</i> attribute of the estimation entry field, 3.
<b>Note:</b> Reverse highlighted numbers and letters are keyed to Figure 107 on page 288.	

You can now save your part and generate its code. From the Visual Builder window, select **Save and generate** → **Part source** in the *File* pull-down menu.

## ASimulMortgageFrameView

All the complexity of ASimulMortgageFrameView is encapsulated in ASimulMortgageView which is connected to AMortgageCalculator. You have almost no connection to add to ASimulMortgageFrameView except one connection from the *buttonClickEvent* event of the commit push-button to the close *action* of the frame window.

You can then save the part and generate its code.

## APropertySearchResultView

APropertySearchResultView displays properties according to a buyer's preferences. This view is relatively complex. Therefore, we introduce a useful approach that will help you aggregate several non-visual parts into one nonvisual part.

APropertySearchResultView is responsible for executing a query against the REAL database. The query is performed according to the clause transmitted by APropertySearchParameterView.

A PropertyManager manages the extraction from the database and interacts with a container to display each matching property as a PropertyContainerObject (see Figure 40 on page 100).

The user can execute several actions from the container:

- Update a property:** APropertyUpdateView can be called to update the property selected in the container.
- Delete a property:** ADeleteDialogView can be called to delete the property selected in the container.
- View the property video:** The property video can be played using an IMMDigitalVideo\* part.
- Access the sale transaction subsystem:** The status of the property can be changed from AVAILABLE to PENDING.
- Display the interested buyers:** Buyers whose preferences match the characteristics of the selected property can be displayed.

In the description that follows, we do not cover the interaction of the Property subsystem with the Buyer subsystem and the Sale transaction subsystem. (Look at the application provided with the book to see how the Property subsystem interacts with the other subsystems.) Also, we do not cover the multimedia facility because it does not provide additional information (see “Adding Multimedia Features” on page 253). Rather, we divide the building of APropertySearchResultView into three sections:

1. We show you how to display a selection of properties in a container, using the PropertyManager part (“Selecting Properties from the Database” on page 291).
2. We explain how to simultaneously retrieve the property information from five tables (“Retrieving Information Across Multiple Tables” on page 292).
3. We describe how to use an Object Factory to update property information that has been retrieved (“Using an Object Factory to Update the Database” on page 295).

Each section brings more and more parts and connections onto the free-form surface. Thus, to complete this view, make sure that you follow the instructions in the correct order.

## Selecting Properties from the Database

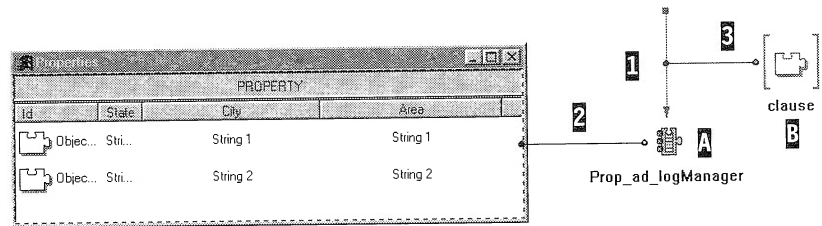
At the design level (Figure 40 on page 100), PropertyManager is responsible for interacting with the database and retrieving properties according to a specific clause. PropertyManager collaborates with PropertyCnr to display the list of properties extracted from the database.

At the implementation level, you decide the type of information to display in the container. To display information from separate tables in the same container, you must build a view that gathers information from each table. As mentioned in Chapter 6, “Mapping Relational Tables Using Data Access Builder,” on page 131, we choose to display three types of information from three different tables:

- ☐ Property characteristics from the PROPERTY table
- ☐ Property address from the PROPERTY\_ADDRESS table
- ☐ Property status from the PROPERTY\_LOG table

Thus, you are provided with the PROP\_AD\_LOG view that joins the PROPERTY, PROPERTY\_ADDRESS, and PROPERTY\_LOG tables. You then map the view, using Data Access Builder, into two parts: Prop\_ad\_logManager and Property\_ad\_log.

You use the select method of Property\_ad\_logManager to run the query against the database. The clause is given as a parameter of the action. The select action is triggered when APropertySearchResultView is ready (Figure 108).



**Figure 108.** Querying the Database

To implement the database query, follow the steps in Table 43 (reverse letters identify the parts, while reverse numbers identify the connections).



**Table 43.** Adding Parts to Query the Database

Step	Action
1	Open APropertySearchResultView part.
2	Add a Prop_ad_logManager* part, <b>A</b> , to the free-form surface.
3	Add an IVBStringPart* variable part, <b>B</b> , to the free-form surface. This part contains the clause transmitted by APropertySearchParameterView.
4	Promote the <b>this</b> attribute of the variable to ensure that the clause is accessible from APropertySearchParameterView.
5	Connect the <b>ready</b> event of APropertySearchResultView to the <b>select</b> action of Prop_ad_logManager, <b>1</b> .
6	Connect the <b>items</b> attribute of the Property_ad_logManager* part to the <b>items</b> attribute of the container, <b>2</b> . This attribute-to-attribute connection keeps the contents of the container synchronized with the set of rows extracted from the database.
7	Connect ( <b>3</b> ) the <b>text</b> attribute of the clause to the <b>clause</b> parameter of the connection <b>1</b> . This connection transmits the contents of the clause to the select method of Prop_ad_logManager.
<b>Note:</b> Reverse highlighted letters and numbers are keyed to Figure 108 on page 291.	

## Retrieving Information Across Multiple Tables

From the container, the user can perform the following actions on a selected property:

**Open** Displays a property and allows the user to update its information.

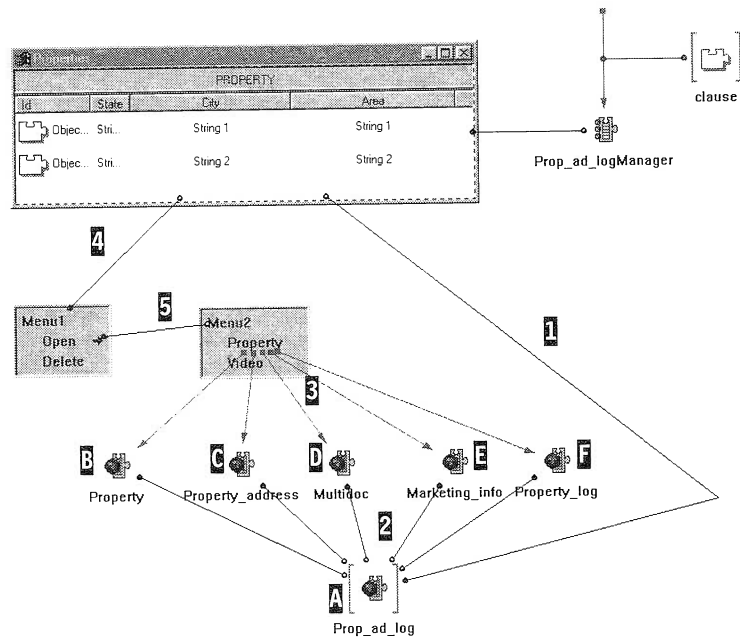
**Delete** Removes a property from the database.

To associate actions with an object selected in the container, you attach a pop-up menu to the container (see “Adding a Pop-up Menu” on page 254).

Each action involves much information. Indeed, updating or deleting a property encompasses updating or deleting records on every table that makes up the property information; that is, PROPERTY, PROPERTY\_ADDRESS, MULTI\_DOC, MARKETING\_INFO, and PROPERTY\_LOG.

To access these tables, you use their associated Data Access Builder parts: Property, Property\_address, Multidoc, Marketing\_info, and Property\_log.

When the user selects a property in the container, its identifier can be propagated to each Data Access Builder part by attribute-to-attribute connections, and the *retrieve* method of each Data Access Builder part can be triggered to retrieve the corresponding record in each table (Figure 109).



**Figure 109.** Retrieving Information from Multiple Tables

To retrieve the property information, follow the steps in Table 44.

Table 44. (Part 1 of 2) Adding Parts to Retrieve Information from Multiple Tables	
Step	Action
1	Add an IVBVariablePart* part, <b>A</b> , to the free-form surface and change its type to Prop_ad_log*. This variable part references the property selected in the container.
2	Add a Property* part, <b>B</b> , on the free-form surface. This part is used to retrieve the Property (characteristics) information of the selected property.

**Table 44.** (Part 2 of 2) Adding Parts to Retrieve Information from Multiple Tables

Step	Action
3	Add a Property_address* part, <b>C</b> , to the free-form surface. This part is used to retrieve the Property_address information of the selected property.
4	Add a Multidoc* part, <b>D</b> , to the free-form surface. This part is used to retrieve the Multi_doc information of the selected property.
5	Add a Marketing_info* part, <b>E</b> , to the free-form surface. This part is used to retrieve the Marketing_info information of the selected property.
6	Add a Property_log* part, <b>F</b> , to the free-form surface. This part is used to retrieve the Property_log information of the selected property.
7	Add two IMenu* parts on the free-form surface.
8	Add three IMenuItem* parts on each IMenu* and change their labels as shown in Figure 109 on page 293.
<b>Note:</b> Reverse highlighted letters are keyed to Figure 109 on page 293.	

Once you have placed the parts on the free-form surface, connect them following the steps in Table 45.

**Table 45.** (Part 1 of 2) Connecting Parts to Retrieve Property Information

Key	Connection	Description
<b>1</b>	selectedElement → this .	Property_ad_log is a placeholder for the property selected in the container.
<b>2</b>	<input type="checkbox"/> property_id → property_id (from Property) <input type="checkbox"/> property_id → address_id (from Property_address) <input type="checkbox"/> property_id → multidoc_id (from Multidoc) <input type="checkbox"/> property_id → property_id (from Marketing_info) <input type="checkbox"/> property_id → property_id (from Property_log)	Synchronize the value of each identifier for the retrieval. These identifiers are selected as <b>Data identifiers</b> in the notebook settings of each relational table (see Chapter 6, “Mapping Relational Tables Using Data Access Builder,” on page 131).
<b>Note:</b> You must draw the connections of Step 2 before you draw the connection of Step 3.		

**Table 45.** (Part 2 of 2) Connecting Parts to Retrieve Property Information

Key	Connection	Description
<b>3</b>	<input type="checkbox"/> commandEvent → retrieve (from Property) <input type="checkbox"/> commandEvent → retrieve (from Property_address) <input type="checkbox"/> commandEvent → retrieve (from Multidoc) <input type="checkbox"/> commandEvent → retrieve (from Marketing_info) <input type="checkbox"/> commandEvent → retrieve (from Property_log)	Retrieve each row according to the identifier value.
<b>4</b>	menu → this	Attach Menu1 to the container
<b>5</b>	menu → this	Attach Menu2 as a submenu of the Open menu item.
<b>Note:</b> Reverse highlighted numbers are keyed to Figure 109 on page 293. To keep the drawings simple, we do not key all connections.		

## Using an Object Factory to Update the Database

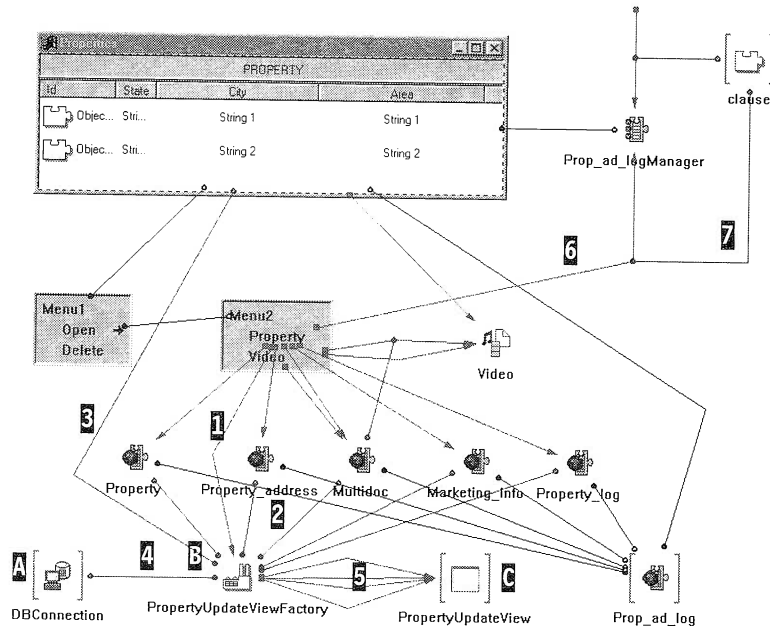
Once the information is retrieved, it can be updated. To update the property selected in the container, APropertySearchResultView collaborates with APropertyUpdateView. The collaboration is based on a “use” relationship: APropertySearchResultView uses the APropertyUpdateView part to provide an updating function from its container. APropertySearchResultView activates the collaboration by creating an instance of APropertyUpdateView. The property information is the link attribute between the two parts. It is transmitted during the instance creation (Figure 110).

To create an instance of a part, use an IVBFactory part (this part is available from the palette in the *Models* category). Factory parts enable your application to dynamically create a visual or a nonvisual part. This differs from parts that are added to the free-form surface and created statically when the application starts.

Like variable parts, factory parts are placeholders for other parts. Each factory part must be set to the type of the class it represents. The factory part works in tandem with a variable part that represents the instance created. You have to use variable parts with factory parts each time you access the attributes or activate a method of the instance created. Unlike variable parts, factory parts run a *create* method, which in turn runs the corresponding part class constructor, creating a new instance.

You can set up the instance attributes during the *create* call in two ways:

- ❑ You provide the factory part with the required attribute values, using parameter connections. In this case, the connections are triggered within the *create* method but after executing the part constructor. The factory creates as many different instances as the attribute values supplied by the parameter connections (see connection 2 in Table 46 on page 297).
- ❑ You set the attribute values in the settings notebook of the factory. In this case, the parameters are passed to the class constructor. The factory creates a clone of the same class each time the **new** action is triggered.



**Figure 110.** Updating a Property

Using factory parts involves the four steps:

1. Adding and setting the factory part
2. Adding and setting the variable part
3. Connecting the event to the factory part
4. Connecting the factory part to its variable part

Table 46 presents step-by-step instructions for using a factory of APropertyUpdateView to update the property information.

<b>Table 46.</b> Updating the Database	
<b>Step</b>	<b>Action</b>
1	Add an IVBVariablePart* part, <b>A</b> , to the free-form surface and change its type to IDatastore*. This variable represents the database connection. It is transmitted to APropertyUpdateView to commit the database transaction.
2	Promote the <b>this</b> attribute of the IVBVariablePart* part <b>A</b> . The IDatastore variable need to be promoted in order to receive the database connection from the caller view: APropertySearchParameterView
3	Add an IVBFactory* part, <b>B</b> , to the free-form surface and change its type to APropertyUpdateView*. This factory object enables creation of an instance of APropertyUpdateView dynamically.
4	Add an IVBVariablePart* part, <b>C</b> , to the free-form surface and change its type to APropertyUpdateView*. This variable represents the APropertyUpdateView instance.
<b>Note:</b> Reverse highlighted letters are keyed to Figure 110 on page 296.	

Once you have placed the parts on the free-form surface, connect them as in Table 47.

<b>Table 47.</b> (Part 1 of 2) Connecting Parts to Update Property Information		
<b>Key</b>	<b>Connection</b>	<b>Description</b>
<b>1</b>	commandEvent → new	Create an instance of APropertyUpdateView.
<b>2</b>	<input type="checkbox"/> property → this <input type="checkbox"/> property_address → this <input type="checkbox"/> multidoc → this <input type="checkbox"/> marketing_info → this <input type="checkbox"/> property_log → this	Transmit each part as a parameter to the factory for update.
<b>3</b>	owner → this	APropertyUpdateView is shown modally, and the frame window is its owner.
<b>4</b>	this → dBConnection	Transmit the database connection to APropertyUpdateView. dBConnection is a variable promoted in APropertyUpdateView.

**Table 47.** (Part 2 of 2) Connecting Parts to Update Property Information

Key	Connection	Description
<b>Note:</b> The order of the next five connections is crucial!		
<b>5</b>	<input type="checkbox"/> newEvent → this <input type="checkbox"/> newEvent → setFocus <input type="checkbox"/> newEvent → showModally <input type="checkbox"/> newEvent → deleteTarget	Associate the instance with the factory. APropertyUpdateView is shown modally. The window instance is deleted when it closes.
<b>6</b>	commandEvent → select	Refreshes the container after the update.
<b>7</b>	Connect the <b>this</b> attribute of the variable to the <b>clause</b> parameter of the connection <b>6</b> .	Clause is given as the parameter for the connection.
<b>Note:</b> Reverse highlighted letters and numbers are keyed to Figure 110 on page 296. To keep the drawings simple, we do not key all connections.		

Ensure that the event-to-action connections from the menu item and APropertyUpdateView instance are in the proper order. To check the order of connections drawn from a part, use the **Reorder Connections...** option from its pop-up menu.

## Deleting a Property

In this section we describe how to implement the delete option of the container pop-up menu (see Figure 111). First we list the parts involved, then we list all of the connections between the parts. For clarity, we do not represent all of the parts and connections that have been added and drawn in the preceding figures.



---

**Note:** Reverse highlighted letters are keyed to Figure 111. To keep the drawings simple, we do not key all connections.

\_\_\_\_\_



When the subparts are in place, connect them to implement the logic of the delete option (see Table 49).

<b>Table 49.</b> (Part 1 of 2) Connecting Parts to Delete a Property from the Database		
<b>Key</b>	<b>Connection</b>	<b>Description</b>
<b>1</b>	selectedElement → this	Property-ad-log is a placeholder for the property selected in the container. Note that this connection has already been drawn in Table 44 on page 293 (see connection <b>1</b> ).
<b>Note:</b> The order of the next two steps is crucial!		
<b>2</b>	<input type="checkbox"/> commandEvent → retrieve (from Property) <input type="checkbox"/> commandEvent → retrieve (from Property_address) <input type="checkbox"/> commandEvent → retrieve (from Multidoc) <input type="checkbox"/> commandEvent → retrieve (from Marketing_info) <input type="checkbox"/> commandEvent → retrieve (from Property_log)	Retrieve each row according to the identifier value.
<b>3</b>	commandEvent → new	Create an instance of ADeleteDialog.
<b>Note:</b> The order of the next two steps is crucial!		
<b>4</b>	<input type="checkbox"/> newEvent → this <input type="checkbox"/> newEvent → setFocus <input type="checkbox"/> newEvent → showModally <input type="checkbox"/> newEvent → deleteTarget	Associate the instance with the factory. ADeleteDialog is shown modally. The window instance is deleted when it closes.
<b>5</b>	newEvent → select	Select action is run to refresh the container after the delete.
<b>6</b>	this → clause	The clause is given as the parameter for the connection.
<b>7</b>	property_id → recordIDText	Transmit the record identifier to the delete dialog window.
<b>Note:</b> The order of the next two steps is crucial!		

**Table 49.** (Part 2 of 2) Connecting Parts to Delete a Property from the Database

Key	Connection	Description
<b>3</b>	<input type="checkbox"/> PushButtonOKButtonClick-Event → delete (from Property) <input type="checkbox"/> PushButttonOKButton-ClickEvent → delete (from Property_address) <input type="checkbox"/> PushButtonOKButtonClick-Event → delete (from Multidoc) <input type="checkbox"/> PushButtonOKButtonClick-Event → delete (from Marketing_info) <input type="checkbox"/> PushButtonOKButtonClick-Event → delete (from Property_log)	Delete each record in each table.
<b>9</b>	PushButtonOKButtonClick-Event → commit	Commit the transaction when done.
<b>Note:</b> Reverse highlighted letters and numbers are keyed to Figure 111 on page 299. To keep the drawings simple, we do not key all connections.		

Ensure that the event-to-action connections from the menu item and the push button of APropertyUpdateView are in the proper order. To check the order, use the **Reorder Connections...** option from the part selected.

You can now save your part and generate its code. From the Visual Builder window, select **Save and generate** → **Part source** in the **File** pull-down menu.

As you may notice, the view is getting a bit confusing, and the parts are cluttered with too many connections. You can hide connections between parts by using the **Browse Connections** option from every part's menu. With this option, you can control the visibility of connections from and to the selected part. If you use the **Browse Connections** option from the free-form surface, you can hide all of the connections of the Composition Editor.

In our view, it is quite easy to identify some nonvisual parts that work exclusively together and could be gathered into one part. In the next section, we show you how to aggregate several nonvisual parts to simplify the view.

# APropertyDelete

APropertyDelete is a nonvisual part created to encapsulate the operations necessary to delete a property from the database. To create this aggregate part, use the Composition Editor and build its logic visually (see Figure 112).

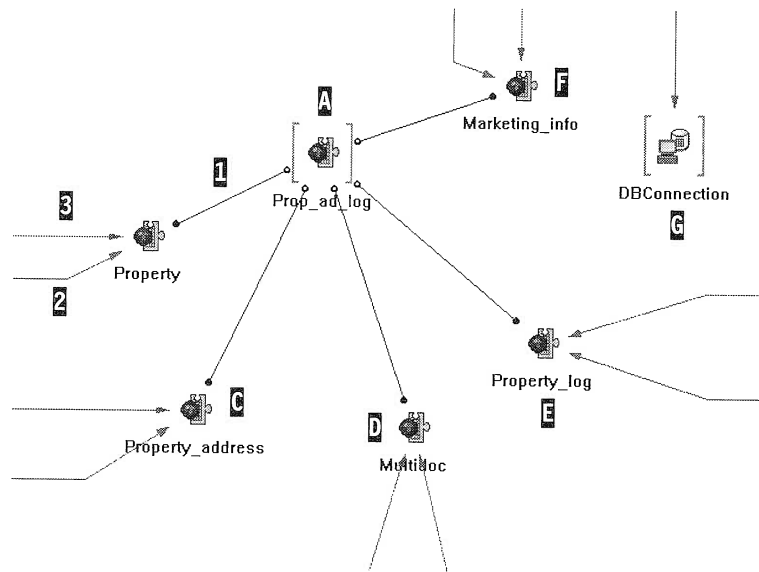


Figure 112. Building APropertyDelete

## Using the Composition Editor to Build a Nonvisual Part

Follow the steps in Table 50 to build the nonvisual part and its components.

Table 50. (Part 1 of 2) Adding Parts to Build APropertyDelete													
Step	Action												
1	<div>From the Visual Builder window, create a nonvisual part as follows:</div> <table><tr><th>Field</th><th>Value</th></tr><tr><td>Class name</td><td>APropertyDelete</td></tr><tr><td>Description</td><td>Nonvisual part to delete properties</td></tr><tr><td>File name</td><td>VRPROP.VBB</td></tr><tr><td>Part type</td><td>Nonvisual part</td></tr><tr><td>Base class</td><td>IStandardNotifier</td></tr></table> <div>The Part Interface Editor is displayed.</div>	Field	Value	Class name	APropertyDelete	Description	Nonvisual part to delete properties	File name	VRPROP.VBB	Part type	Nonvisual part	Base class	IStandardNotifier
Field	Value												
Class name	APropertyDelete												
Description	Nonvisual part to delete properties												
File name	VRPROP.VBB												
Part type	Nonvisual part												
Base class	IStandardNotifier												

**Table 50.** (Part 2 of 2) Adding Parts to Build APropertyDelete

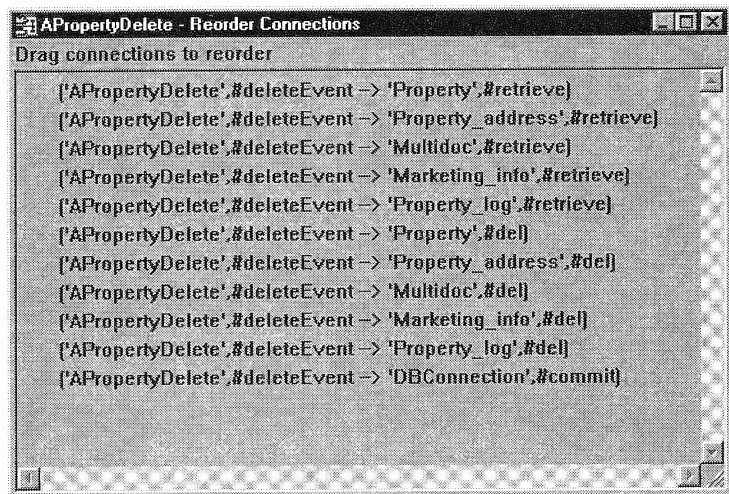
Step	Action
2	From the Event page of the Part Interface Editor, create a new event, <b>deleteEvent</b> , using the default settings. This event triggers the <b>delete</b> action over the five tables. To send this event to the subparts that make up APropertyDelete, you create, in Step 3, an action that sends a deleteEvent notification to each subpart. When notified, each subpart executes the <b>delete</b> action.
3	From the Action page of the Part Interface Editor, create a new action, <b>delete</b> , using the default settings. This action will be called from the commandEvent of APropertySearchResultView.
4	Switch to the Composition Editor.
5	Add an IVBVariablePart* part to the free-form surface and change its type to Prop_ad_log*, <b>A</b> . This variable receives its value from the Prop_ad_log variable part of APropertySearchResultView and must be promoted.
6	Promote the <b>this</b> attribute of the Prop_ad_log* variable part.
7	Add a Property* part to the free-form surface, <b>B</b> .
8	Add a Property_address* part to the free-form surface, <b>C</b> .
9	Add a Multidoc* part to the free-form surface, <b>D</b> .
10	Add a Property_log* part to the free-form surface, <b>E</b> .
11	Add a Marketing_info* part to the free-form surface, <b>F</b> .
12	Add an IVBVariablePart* part to the free-form surface, <b>G</b> , and change its type to IDatastore*. This variable receives the database connection from APropertySearchResultView and must be promoted.
13	Promote the <b>this</b> attribute of the <b>IDatastore*</b> variable part.
<b>Note:</b> Reverse highlighted letters are keyed to Figure 112 on page 302.	

To connect the parts to each other, refer to the original view, APropertySearchResultView, and to Table 51.

**Table 51.** Connecting Parts to Build APropertyDelete

Key	Connection	Description
<b>1</b>	<input type="checkbox"/> property_id → property_id (from Property) <input type="checkbox"/> property_id → address_id (from Property_address) <input type="checkbox"/> property_id → multidoc_id (from Multidoc) <input type="checkbox"/> property_id → property_id (from Marketing_info) <input type="checkbox"/> property_id → property_id (from Property_log)	Synchronize the value of each identifier for the retrieval (only the first connection is keyed in Figure 112 on page 302).
<b>2</b>	<input type="checkbox"/> deleteEvent → retrieve (from Property) <input type="checkbox"/> deleteEvent → retrieve (from Property_address) <input type="checkbox"/> deleteEvent → retrieve (from Multidoc) <input type="checkbox"/> deleteEvent → retrieve (from Marketing_info) <input type="checkbox"/> deleteEvent → retrieve (from Property_log)	Retrieve each record from each table (only the first connection is keyed in Figure 112 on page 302).
<b>3</b>	<input type="checkbox"/> deleteEvent → del (from Property) <input type="checkbox"/> deleteEvent → del (from Property_address) <input type="checkbox"/> deleteEvent → del (from Multidoc) <input type="checkbox"/> deleteEvent → del (from Marketing_info) <input type="checkbox"/> deleteEvent → del (from Property_log)	Delete the property across the five tables (only the first connection is keyed in Figure 112 on page 302).
<b>4</b>	deleteEvent → commit	Commit the transaction in the database.
<b>Note:</b> Reverse highlighted numbers are keyed to Figure 112 on page 302.		

Ensure that the event-to-action connections used in APropertyDelete are in the proper order. To check the order, use the **Reorder Connections...** option from the free-form surface (see Figure 113).

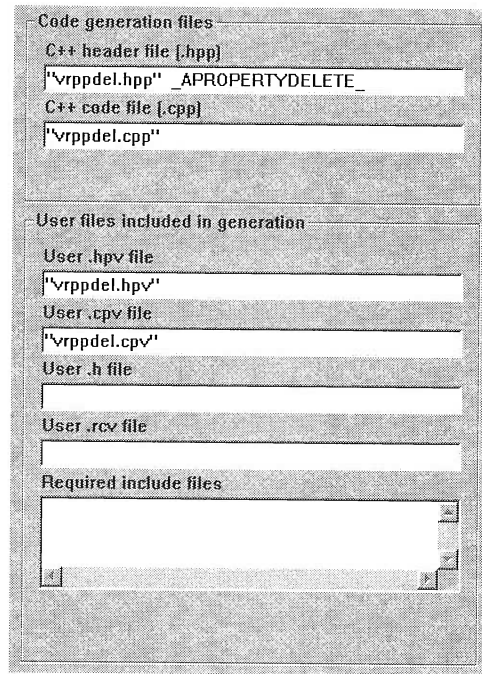


**Figure 113.** Order of Connections for APropertyDelete

Notice that the sequence of the part's logic is directly expressed in this window. You can follow, line by line, which statement is executed to delete each part of the property information in the five relational tables.

Before saving and generating the code, switch to the Class Editor. In the *User files included in generation* group box, type in the following file names (see Figure 114):

- ☐ User .hvp file: **vrppdel.hvp**
- ☐ User .cpv file: **vrppdel.hvp**



**Figure 114.** Detail of the Class Editor

You can save and generate the part's source and the part's features. Then, from your favorite editor, edit the `vrppdel.cpv` file and add the code in Figure 115 to the delete method:

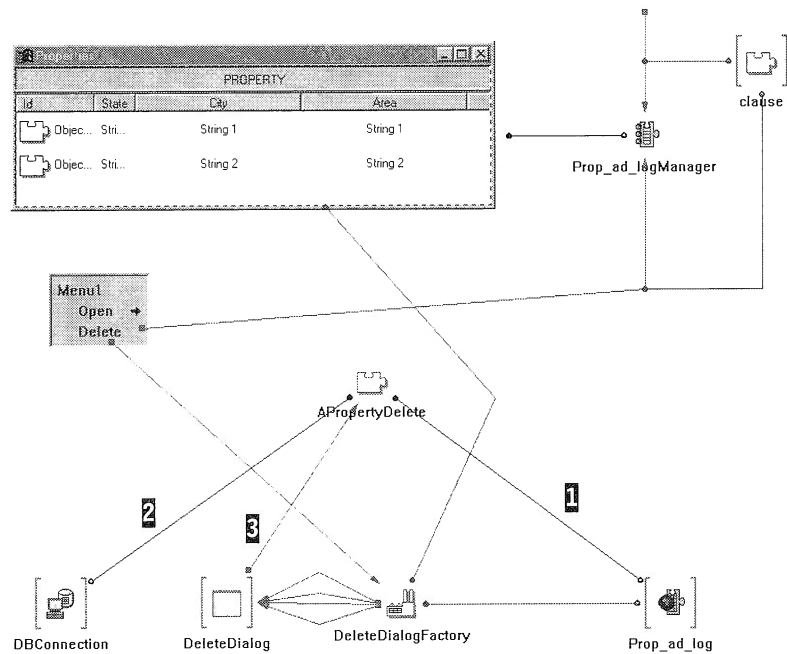
```
notifyObservers(INotificationEvent(APropertyDelete::deleteEventId, *this))
```

**Figure 115.** Code to Generate deleteEventId

When `notifyObservers` is called, it notifies the event-to-action connections of `APropertyDelete`, which in turn triggers the **retrieve**, **delete**, and **commit** actions on the database.

To use `APropertyDelete`, just drop it on the free-form surface of `APropertySearchResultView` and connect the parts as follows (see Figure 116):

1. Connect the **this** attribute of the `Prop_ad_log` variable to the `propAdLog` promoted variable part of `APropertyDelete`, **1**.
2. Connect the **this** attribute of the `DBConnection` variable part to the `dbConnection` promoted variable part of `APropertyDelete`, **2**.
3. Connect the **pushButtonOKButtonClickEvent** event of the `ADeleteDialog` variable to the **del** action of `APropertyDelete`, **3**.



**Figure 116.** Using APropertyDelete Part

Notice that the Data Access Builder parts have been removed (they are still necessary for the update function). You can embed ADeleteDialogView factory and its instance in APropertyDelete part to encapsulate the entire process. You just need a connection from the **commandEvent** of the delete menu item to the **delete** action of APropertyDelete.

## APropertySearchParameterView

APropertySearchParameterView is used to build a query on property criteria. The query is then sent to APropertySearchResultView to be executed against the database.

The query is the link attribute between APropertySearchParameterView and APropertySearchResultView and can be traced in the design object model of the application (see Figure 40 on page 100).

When the user clicks on the **Search** push button in APropertySearchParameterView, the query is sent as a clause to APropertySearchResultView. APropertySearchResultView collaborates with a



Data Access Builder part, PropertyManager, to execute the query against the database. Upon completion, the container held by APropertySearchResultView is updated with the matching properties.

The query is built according to the following property criteria:

- ☐ Area
- ☐ Price range
- ☐ Size range
- ☐ Number of bedrooms
- ☐ Number of bathrooms

The user can choose to search a property by using all or some of the criteria. The user can choose each criterion involved in the query by selecting its corresponding check box. After selecting a check box, the user can specify a value for the criterion.

The *Property Search* use case is extended by calculating the highest mortgage the buyer can afford. As mentioned in “Requirement Specifications” on page 62, the simulation is activated when the buyer searches for some affordable properties and the price range is not provided.

Because APropertySearchParameterView includes many connections, you build it in three phases:

- ☐ In the first phase (“Managing the User Input” on page 308), you use check box controls and IVBBooleanPart parts to control the user input.
- ☐ In the second phase (“Adding The Mortgage Calculation” on page 310), you add the mortgage calculation facility, which provides a price range for building the clause, and use a custom logic connection to decide upon its activation.
- ☐ In the last phase (“Building the Clause” on page 311), you build the clause from the user input by using an event-to-member function connection.

## Managing the User Input

You manage the user input by using a check box control as a switch. This switch enables or disables user access to the criteria entry fields. To react to the selection of the check box, you use an IVBBooleanPart part that can be found in the VBSAMPLE.VBB file. This part holds a Boolean value and can react according to its value (for more information, consult the *Visual Builder Parts Reference*).

Two events are relevant:

- valueFalseEvent** Enables the part to notify other parts as soon as its value is set to FALSE
- valueTrueEvent** Enables the part to notify other parts as soon as its value is set to TRUE

The check box, in turn, is a two-status control that can be either selected or unselected. If it is selected, its *selected* attribute holds a TRUE value. If it is not selected, it holds a FALSE value.

The idea is now quite simple. We use the criterion number of bedrooms to illustrate the description (see Figure 117). The same building process applies to the other criteria. Here are the steps:

1. Connect the *selected* attribute of the check box (A) to the *value* attribute of IVBBooleanPart (B) with an attribute-to-attribute connection (1).
2. Connect the **valueFalseEvent** event of IVBBooleanPart to the **disable** action of the entry field (C) to prevent the user from typing in a value (2).
3. Connect the **valueTrueEvent** event of IVBBooleanPart to the **enable** action of the entry field to enable the user to enter the number of bedrooms (3).
4. Connect the **valueTrueEvent** event of IVBBooleanPart to the **setFocus** action of the entry field to position the cursor (4).

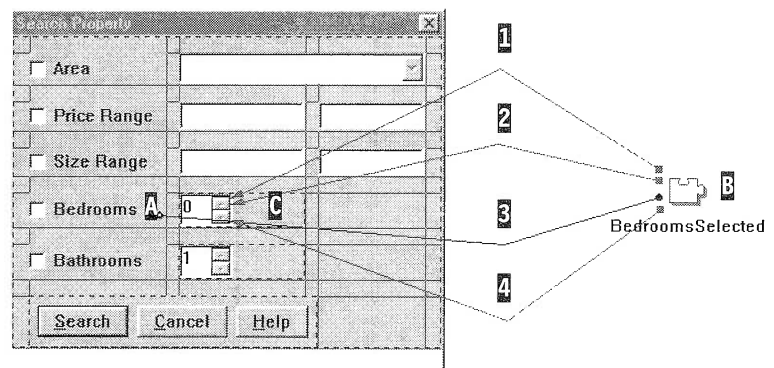
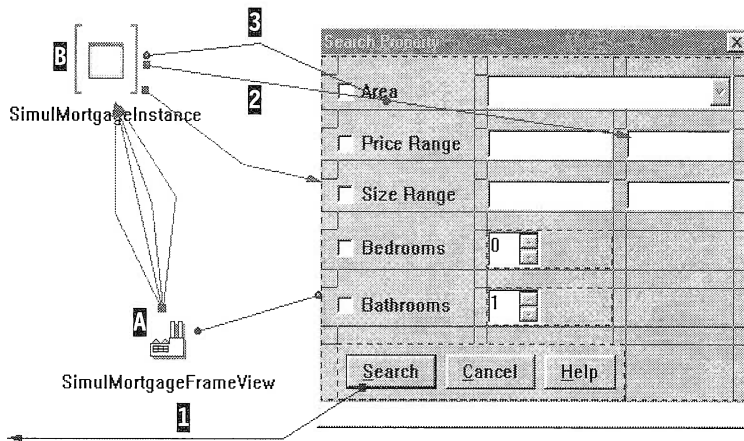


Figure 117. Number of Bedrooms Selection

When the application starts, the input controls (entry fields, drop-down list box, and spin buttons) are disabled because their *enabled* check box is deselected on the Control page of their settings notebook (see “APropertySearchParameterView” on page 200).

## Adding The Mortgage Calculation

The mortgage calculation is activated when the *price range* check box is selected and neither the minimum price, nor the maximum price is provided. It is rather complex to implement such a logical condition using IVBBooleanPart parts. A rational solution here consists of using a custom logic connection **1** (see Figure 118) which fires the **create** method of the IVBFactory part **A** (see “Using an Object Factory to Update the Database” on page 295) to produce an instance of ASimulMortgageFrameView, **B**, when the condition is fulfilled. The highest estimated mortgage is returned to set the value of the maximum price entry field. The entry-field content is set when the *closeEvent* of SimulMortgageInstance occurs (see **2**). The highest mortgage value, promoted as *estimation*, is transmitted to the connection **2** by the parameter connection **3**.



**Figure 118.** Activating The Mortgage Calculation

The target of the custom logic connection must be the free-form surface which represents APropertySearchParameterView. From it, you can access all the subparts and check their status as shown in Figure 115.

```

if ( target→iCheckBoxPrice→isSelected() &&
    target→iMinimumPrice→isEmpty() &&
    target→iMaximumPrice→isEmpty() )
    target→iSimulMortgageFrameView→create();

```

**Figure 119.** Code for Activating the Mortgage Simulation

## Building the Clause

The query clause is built from the contents of the input controls whose corresponding check box is selected.

You use an IVBStringPart part to represent the clause. This part can be found in the VBSAMPLE.VBB file. The part is initialized with an event-to-member function connection (see “Using a Member Function Connection” on page 283 and connection 6 in Figure 122 on page 315), which triggers a member function of APropertySearchParameterView: *BuildClause*. The member function accesses:

- ❑ The check boxes to verify that they are selected
- ❑ The entry fields related to the selecte check boxes
- ❑ The Clause part to set its contents

Because BuildClause is a member function of APropertySearchParameterView, each subpart can be accessed from BuildClause by its name prefixed with an *i* (see Figure 120).

```

Boolean addAnd = false;
target → iClause → assignTextToEmpty();

/* area is selected */
if( iAreaSelected → value() ) {
    iClause → appendText("area = " + iArea → text() + "");
    addAnd = true;
}

/* bedrooms is selected */
if( iBedroomsSelected → value() )
    if( addAnd )
        iClause → appendText(" and bedrooms = " +
            IString(iBedrooms → value()));
    else {
        iClause → appendText("bedrooms = " +
            IString(iBedrooms → value()));
        addAnd = true;
    }
}

/* price is selected */
.....
if (addAnd)
    iClause → appendText(" and status = 'AVAILABLE'");
else
    iClause → appendText("status = 'AVAILABLE'");

return addAnd;

```

**Figure 120.** Code Fragment of BuildClause Member Function

The BuildClause member function is declared as a public member function with the following prototype: Boolean BuildClause(). The return code enables you to check whether the clause is empty (see “Using a Message Box to Display the Clause” on page 318). We could have declared the member function with input parameters, such as the price range or the size range, and transmitted the parameter values through parameter connections from the entry controls to the event-to-member function connection. The member function is declared in the vrpsrcv.hpv file and defined in the vrpsrcv.cpv file. These two files must be added to the definition of APropertySearchParameterView by filling in the corresponding entry fields in the *User files included in generation* group box (see “Using a Member Function Connection” on page 283).

The complete code of the BuildClause member function is given in Appendix E on page 525.

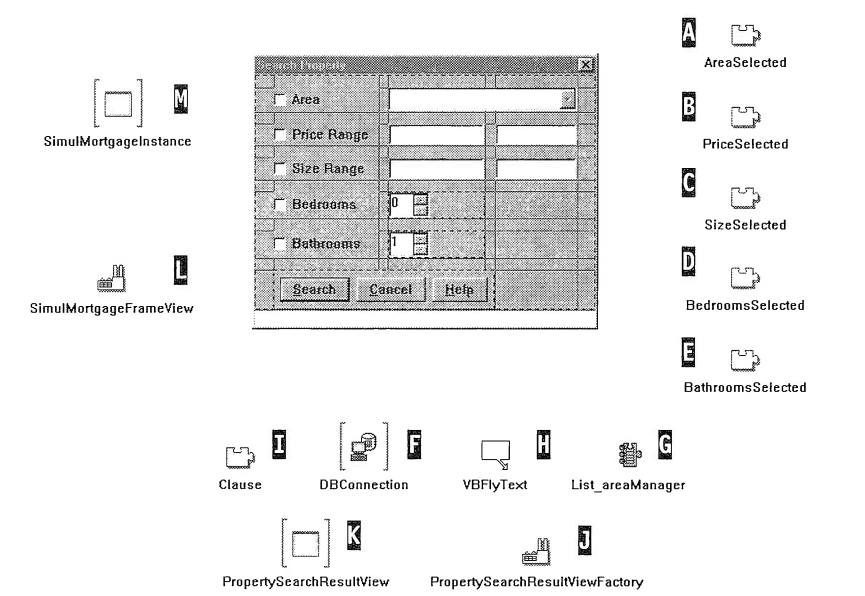
Notice that the code remains simple and does not include all controls that would be necessary for a real application.

APropertySearchParameterView collaborates with APropertySearchResultView to execute the query against the database. As with the relationship between APropertySearchResultView and AProper-

tyUpdateView or APropertyDeleteView, the collaboration is based on a use relationship: APropertySearchParameterView uses the SELECT properties service of APropertySearchResultView. APropertySearchParameterView activates the collaboration by creating an instance of APropertySearchResultView. The clause is the link attribute between these two parts. It is transmitted during the instance creation (see connection **12** in Figure 122 on page 315).

To create an instance of APropertySearchResultView, use an IVB-Factory part (see “Using an Object Factory to Update the Database” on page 295).

To refine the view, add the subparts required as shown in Figure 121 and laid out in Table 52.



**Figure 121.** Subparts of APropertySearchParameterView

Table 52. (Part 1 of 3) Adding Parts to Build APropertySearchParameterView	
Step	Action
1	Open the APropertySearchParameterView part.

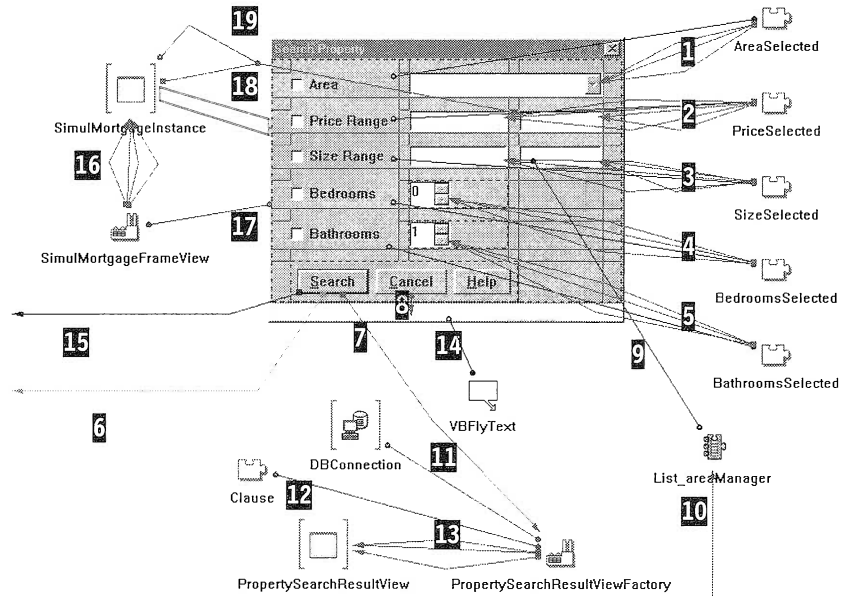
**Table 52.** (Part 2 of 3) Adding Parts to Build  
APropertySearchParameterView

Step	Action
2	<p>Add five IVBBooleanPart* parts on the free-form surface and change their names as follows:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> AreaSelected <b>A</b></li> <li><input type="checkbox"/> PriceSelected <b>B</b></li> <li><input type="checkbox"/> SizeSelected <b>C</b></li> <li><input type="checkbox"/> BedroomsSelected <b>D</b></li> <li><input type="checkbox"/> BathroomsSelected <b>E</b></li> </ul>
3	Add an IDatastore* variable part, <b>F</b> , to the free-form surface. This variable is transmitted to the factory of APropertySearchResult-View.
4	Add a List_areaManager* part, <b>G</b> . This part is used to list the area available from the database, in the area drop-down list box (use <i>Option</i> → <i>Add Part...</i> from the Composition Editor).
5	Add an IVBFlyText* part, <b>H</b> , to the free-form surface. This part is used to display the fly-over long-text help in the window info area.
6	Add short and long fly-over help texts to the check boxes, entry fields, and push buttons.
7	Add an IVBStringPart* part, <b>I</b> , to the free-form surface. This part holds the clause built by the custom logic connection.
8	Add an IVBFactory* part, <b>J</b> , to the free-form surface. This part creates an instance of APropertySearchResultView.
9	Change the factory type to <b>APropertySearchResultView*</b> .
10	Check the <b>Auto delete</b> mark on the General page of the factory notebook settings. The window instance of APropertySearchResult-View is deleted automatically when it closes. The Auto delete check mark must be checked only for visual part instances that are shown modelessly.
11	Add an APropertySearchResultView* variable part, <b>K</b> , to the free-form surface. This part represents the instance created by the factory.
12	Add an IVBFactory* part, <b>L</b> , to the free-form surface. This part creates an instance of ASimulMortgageFrameView.
13	Change the factory type to <b>ASimulMortgageFrameView*</b> .
14	Add an ASimulMortgageFrameView* variable part, <b>M</b> , to the free-form surface. This part represents the instance created by the factory. This view is shown modally.

**Table 52.** (Part 3 of 3) Adding Parts to Build  
APropertySearchParameterView

Step	Action
<b>Note:</b> Reverse highlighted numbers are keyed to Figure 121 on page 313.	

Once you have placed the subparts on the free-form surface, you can draw the connections as shown in Figure 122 and Table 53.



**Figure 122.** APropertySearchParameterView: The Big Picture

**Table 53.** (Part 1 of 4) Connecting Parts to Build  
APropertySearchParameterView

Key	Connection	Description
<b>1</b>	<input type="checkbox"/> selected (checkBoxArea) → value (AreaSelected) <input type="checkbox"/> valueTrueEvent → enable <input type="checkbox"/> valueTrueEvent → setFocus <input type="checkbox"/> valueFalseEvent → disable	Control the input for the area.



**Table 53.** (Part 2 of 4) Connecting Parts to Build  
APropertySearchParameterView

Key	Connection	Description
<b>2</b>	<input type="checkbox"/> selected (CheckBoxPrice) → value (PriceSelected) <input type="checkbox"/> valueTrueEvent → enable (MinimumPrice) <input type="checkbox"/> valueTrueEvent → enable (MaximumPrice) <input type="checkbox"/> valueTrueEvent → setFocus (MinimumPrice) <input type="checkbox"/> valueFalseEvent → disable (MinimumPrice) <input type="checkbox"/> valueFalseEvent → disable (MaximumPrice)	Control the input for the price range.
<b>3</b>	<input type="checkbox"/> selected (CheckBoxSize) → value (SizeSelected) <input type="checkbox"/> valueTrueEvent → enable (MinimumSize) <input type="checkbox"/> valueTrueEvent → enable (MaximumSize) <input type="checkbox"/> valueTrueEvent → setFocus (MinimumSize) <input type="checkbox"/> valueFalseEvent → disable (MinimumSize) <input type="checkbox"/> valueFalseEvent → disable (MaximumSize)	Control the input for the size range.
<b>4</b>	<input type="checkbox"/> selected (CheckBoxBedrooms) → value (Bedrooms-Selected) <input type="checkbox"/> valueTrueEvent → enable <input type="checkbox"/> valueTrueEvent → setFocus <input type="checkbox"/> valueFalseEvent → disable	Control the input for the number of bedrooms.
<b>5</b>	<input type="checkbox"/> selected (CheckBoxBathrooms) → value (Bathrooms-Selected) <input type="checkbox"/> valueTrueEvent → enable <input type="checkbox"/> valueTrueEvent → setFocus <input type="checkbox"/> valueFalseEvent → disable	Control the input for the number of bathrooms.
<b>Note:</b> The order of the next two connections is crucial!		
<b>6</b>	buttonClickEvent → Build-Clause()	Build the clause and initialize the IVBStringPart.
<b>7</b>	buttonClickEvent → new	Create an instance of AProperty-SearchResultView.

**Table 53.** (Part 3 of 4) Connecting Parts to Build APropertySearchParameterView

Key	Connection	Description
<b>8</b>	buttonClickEvent → close	Close the window if user selects the Cancel option.
<b>9</b>	items → items	Synchronize the records of the relational view with the contents of the drop-down list box.
<b>10</b>	ready → refresh	Load the rows of the List_area table into the IVSequence held by the List_areaManager* part.
<b>11</b>	DBConnection → this	Transmit the database connection to APropertySearchResultView. DBConnection is a variable promoted in APropertySearchResultView.
<b>12</b>	Clause → this	Transmit the clause to APropertySearchResultView. Clause is a variable promoted in APropertySearchResultView.
<b>Note:</b> The order of the next three connections is crucial!		
<b>13</b>	<input type="checkbox"/> newEvent → this <input type="checkbox"/> newEvent → setFocus <input type="checkbox"/> newEvent → visible	Associate the instance with the factory. APropertySearchResultView is shown modelessly.
<b>14</b>	this → longTextControl	Set the target of the fly-over help to the window info area.
<b>15</b>	buttonClickEvent → custom-Logic	Test the activation condition for the Mortgage calculation.
<b>16</b>	<input type="checkbox"/> newEvent → this <input type="checkbox"/> newEvent → setFocus <input type="checkbox"/> newEvent → showModally <input type="checkbox"/> newEvent → deleteTarget	Associate the instance with the factory. SimulMortgageInstance is shown modally.
<b>17</b>	this → owner	Set the owner of SimulMortgageInstance.
<b>18</b>	closeEvent → valueAsDouble	Set maximum price when the instance closes.
<b>19</b>	newValue → estimation	Transmit the estimation as a parameter.

**Table 53.** (Part 4 of 4) Connecting Parts to Build  
APropertySearchParameterView

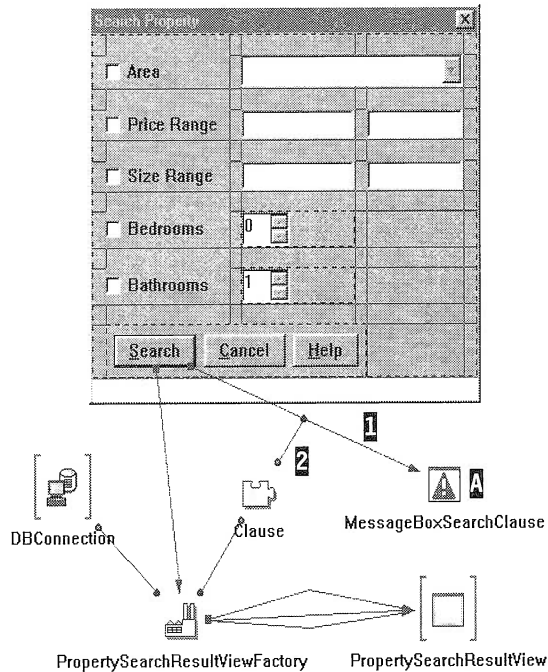
Key	Connection	Description
<b>Note:</b> Reverse highlighted numbers are keyed to Figure 122 on page 315. To keep the drawings simple, we do not key all connections.		

For additional practice, you may want to build a nonvisual part to generate the clause. For example, you can consider a part made of five attributes (area, price, size, bedrooms, bathrooms, and clause). The *clause* attribute is updated dynamically according to the value of the other attributes (see Chapter 8, “Creating Nonvisual Parts,” on page 225 for a similar nonvisual part). To use the part in APropertySearchResultView, connect each **text** attribute from the drop-down list box, entry field, and numeric spin buttons to its corresponding attribute in the nonvisual part and connect the **clause** attribute of the nonvisual part to the *Clause* parameter of PropertySearchResultViewFactory.

You can display the clause before it is used by PropertySearchResultViewFactory. In the next section, we show you how to use a message box to display the query clause.






## Using a Message Box to Display the Clause

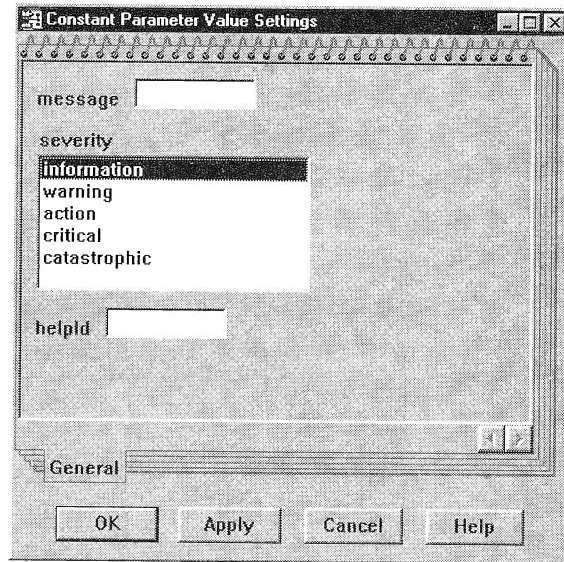
A message box can be a useful tool for debugging your program when things go wrong (see Figure 123). You can use a message box as a substitute for the standard *printf* function to display variable or parameter values at run time. Using the message box and the Boolean part, you can manage the behavior of the box to display all types of messages ranging from informative to critical.



**Figure 123.** Using a Message Box to Display the Clause

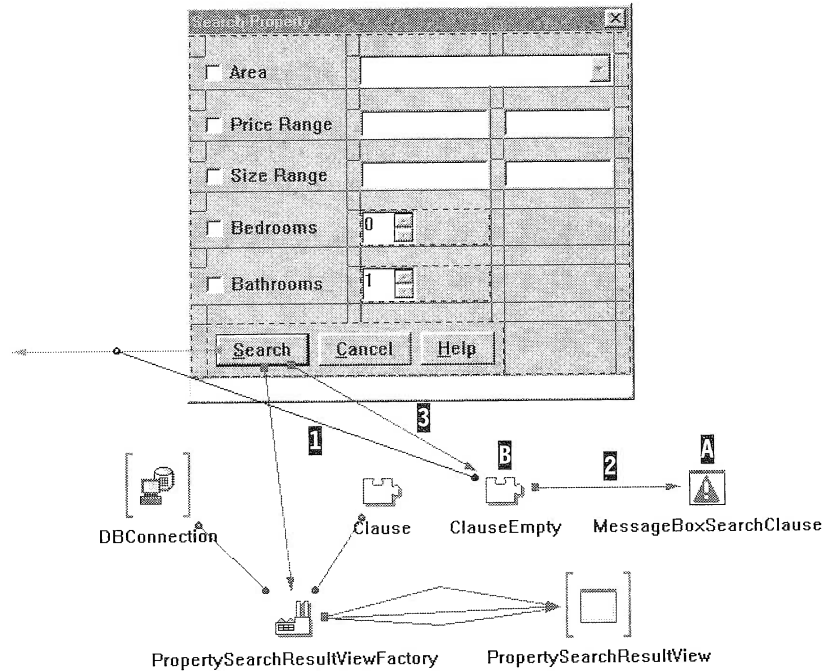
To display the SELECT clause in a simple message box, follow the steps in Table 54.

Table 54. Using a Message Box to Display the Clause	
Step	Action
1	Add an <code>IMessageBox*</code> part,  , to the free-form surface.
2	Connect the <b>buttonClickEvent</b> event of the <i>Search</i> push button to the <b>show</b> action of the message box,  .
3	Connect (see  ) the <b>text</b> attribute of the Clause part to the <b>message</b> parameter of the connection,  .
4	Open the settings of the connection  and click on the <b>Set Parameters...</b> push button. In the dialog box, select <b>information</b> in the severity list box (see Figure 124). The severity parameter associates an icon type with the message box.
<b>Note:</b> Reverse highlighted numbers are keyed to Figure 123.	



**Figure 124.** Message Box Parameter

Let us improve our part by adding the capability to display a warning message each time the clause is empty. In the following example, you use the same message box to display an information message when the clause is not empty and a warning message when the clause is empty (Figure 125).



**Figure 125.** Using Message Box to Display a Warning Message

You use the return code (see Figure 120 on page 312) of the member function connection to decide whether or not the clause is empty:

- ❑ The clause is empty if the return code is equal to FALSE.
- ❑ The clause is not empty if the return code is equal to TRUE.

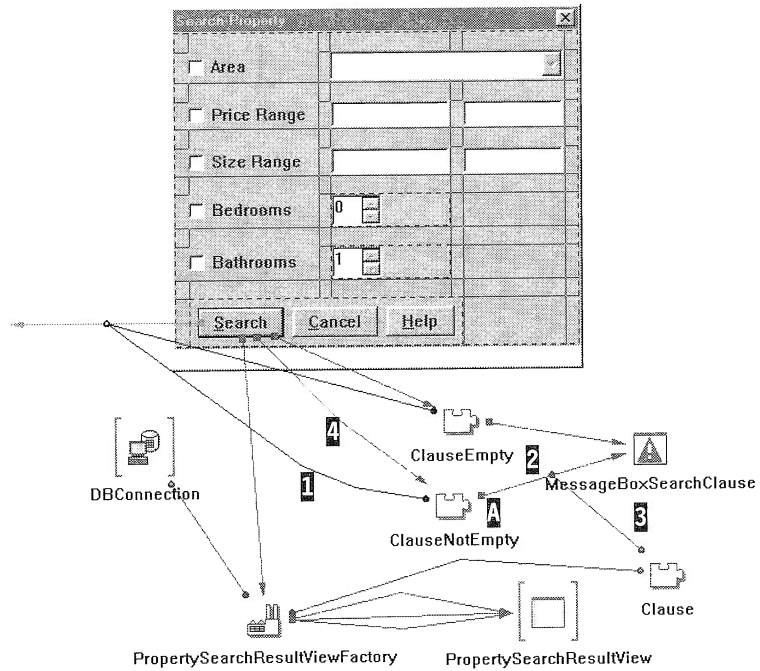
IVBBooleanPart part provides you with the means to trigger the **show** action of the message box. Use IVBBooleanPart to display the warning message, as shown in the step-by-step instructions in Table 55.

Table 55. (Part 1 of 2) Using Message Box to Display a Warning Message	
Step	Action
1	Add an IMessageBox* part, <b>A</b> , to the free-form surface.
2	Add an IVBBooleanPart* part, <b>B</b> , to the free-form surface.
3	Open the part settings of IVBBooleanPart* and set its value to <b>TRUE</b> in selecting the <i>value</i> check box.

**Table 55.** (Part 2 of 2) Using Message Box to Display a Warning Message

Step	Action						
4	Connect the <b>actionResult</b> attribute of the member function connection to the <b>value</b> attribute of the IVBBooleanPart* part, <b>1</b> .						
5	Connect the <b>valueFalseEvent</b> event of the IVBBooleanPart* part to the <b>show</b> action of the message box, <b>2</b> .						
6	Open the settings of the connection <b>2</b> and set the values as follows: <table> <tr> <th>Parameter</th><th>Value</th></tr> <tr> <td>message</td><td>Clause is empty! All records will be retrieved.</td></tr> <tr> <td>severity</td><td>Warning</td></tr> </table> <p>When the clause is empty, the part value of IVBBooleanPart* is set to FALSE and triggers the display of the warning message.</p>	Parameter	Value	message	Clause is empty! All records will be retrieved.	severity	Warning
Parameter	Value						
message	Clause is empty! All records will be retrieved.						
severity	Warning						
7	Connect the <b>buttonClickEvent</b> event of the <i>Search</i> push button to the <b>assignValueToTrue</b> action of the IVBBooleanPart* part, <b>3</b> . This connection is necessary to reset the IVBBooleanPart* part so that it can notify connection <b>2</b> when the clause is empty. In effect, if the part value of IVBBooleanPart* is already FALSE and the <b>Search</b> push button is clicked, connection <b>1</b> does not change the part value of IVBBooleanPart* when it triggers because it is already set to FALSE. Thus, the Boolean part does not notify connection <b>2</b> , and the warning message box does not show again.						
<b>Note:</b> Reverse highlighted letters and numbers are keyed to Figure 125 on page 321.							

Let us now use the same message box to display the clause when it is not empty. The first idea that comes to mind is to reuse the IVBBooleanPart part to trigger the **show** action of the message box whenever the clause is not empty. In effect, you could connect the **valueTrueEvent** event of the IVBBooleanPart part to the **show** action of the message box. Unfortunately, you must reset the IVBBooleanPart part to the FALSE value if you want the message box to show when the clause is not empty twice in succession. Thus, you need another IVBBooleanPart to activate the message box action (Figure 126).



**Figure 126.** Message Box Displaying a Warning or Information Message

To display the clause in a message box when it is not empty, refine the part as shown in Table 56.

Table 56. (Part 1 of 2) Using Message Box to Display a Warning or Information Message	
Step	Action
1	Add another IVBBooleanPart* part, <b>A</b> , to the free-form surface.
2	Open the IVBBooleanPart* part settings and set its value to <b>FALSE</b> in selecting and deselecting the <i>value</i> check box. In effect, to ensure that initial part values are set the way you expect, always set them explicitly. When you drop the IVBBooleanPart* part on the free-form surface, its value is undefined. When you open the settings from the part, the check box for the value is not selected. The deselected check box indicates only that the value has never been set. To set it to <b>FALSE</b> , you must select and deselect the check box.
3	Connect the <i>actionResult</i> attribute of the member function connection to the <i>value</i> attribute of the IVBBooleanPart* part, <b>A</b> .



**Table 56.** (Part 2 of 2) Using Message Box to Display a Warning or Information Message

Step	Action
4	Connect the <b>valueTrueEvent</b> event of the IVBBooleanPart* part to the <b>show</b> action of the message box, 2.
5	Open the settings of connection 2 and set the severity parameter to <b>information</b> .  When the clause is not empty, the part value of IVBBooleanPart* is set to TRUE and triggers the display of the information message.
6	Connect (see 5) the <b>text</b> attribute of the Clause part to the <b>message</b> parameter of connection 2.
7	Connect the <b>buttonClickEvent</b> event of the <i>Search</i> push button to the <b>assignFalseToFalse</b> action of the IVBBooleanPart* part, 4. This connection is necessary to reset the IVBBooleanPart* part so that it can notify connection 2 when the clause is not empty.
<b>Note:</b> Reverse highlighted letter and numbers are keyed to Figure 126 on page 323.	

You can now save your part and generate its code. From the Visual Builder window, select *Save and generate* → *Part source* in the *File* pull-down menu.

## AUploadView

AUploadView is a generic view that enables the user to export tables in a specific directory. A command file is called for this purpose. There is one command file per subsystem:

- ☐ BUYER.BAT command file generates export files for the tables related to the buyer information.
- ☐ PROPERTY.BAT command file generates export files for the tables related to the property information.
- ☐ SALE.BAT command file generates export files for the tables related to the sale transaction information.

In addition, the UPLOAD.BAT command file generates the export files of all relational tables.

The export files are generated in the upload directory. The user sets the directory in two ways:

- ❑ By selecting the settings option from the menu bar of the Visual Realty application main window
- ❑ By selecting the settings option from the menu bar of the service subsystem window

AUploadView is reused in all subsystems. Because a different command file is used for each subsystem, it is the responsibility of each subsystem to provide AUploadView with the correct command file when required. A string part, **B**, is used to hold the command file name for this purpose. The text attribute of this part is promoted. Each subsystem which calls AUploadView initialized this attribute by providing the factory (of type AUploadView\*) with the corresponding value.

The upload directory is retrieved from the application profile setup in ARealSettingsView (see “AResultSettingsView” on page 218 and Figure 127).

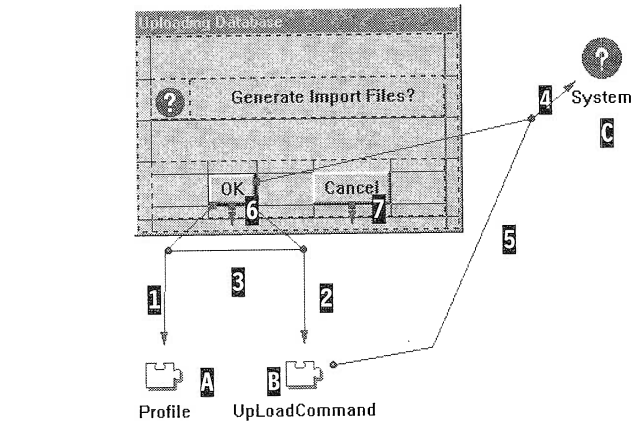


Figure 127. AUploadView

To build the AUploadView part, add the necessary subparts to the free-form surface and make the connections as shown in Table 57.

Table 57. (Part 1 of 3) Building AUploadView Part	
Step	Action
1	Open the AUploadView part.
2	Add an IProfile* part to the free-form surface, <b>A</b> . This part retrieves the upload directory from the application profile.

**Table 57.** (Part 2 of 3) Building AUploadView Part

Step	Action						
3	<p>Open the settings notebook of the IProfile* part and set the fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>defaultApplicationName</td><td>REAL</td></tr> <tr> <td>name</td><td>REAL.INI</td></tr> </table> <p>The application profile is set to REAL.INI, and its default application name is REAL.</p>	Field	Value	defaultApplicationName	REAL	name	REAL.INI
Field	Value						
defaultApplicationName	REAL						
name	REAL.INI						
4	Add an IVBStringPart* variable part to the free-form surface, <b>3</b> , and promote its <i>text</i> attribute. This variable part holds the specific command file name that is to be executed. The file name is transmitted by the calling view.						
5	Add an stdlibSamples* part on the free-form surface, <b>4</b> . This part is located in the VBSAMPLE.VBB file. It provides a wrapper to call your own external functions, such as a command file or external C routines.						
6	Connect the <b>buttonClickEvent</b> event from the <i>OK</i> push button to the action <b>elementWithKey</b> action of the profile part, <b>1</b> . This action retrieves the data associated with a corresponding key for a specific application.						
7	<p>Open the settings of the connection <b>1</b> and set the fields as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>key</td><td>UPLOADPATH</td></tr> <tr> <td>applName</td><td>REAL</td></tr> </table> <p>The upload directory associated with the key UPLOADPATH is retrieved from the REAL application name in the REAL.INI profile.</p>	Field	Value	key	UPLOADPATH	applName	REAL
Field	Value						
key	UPLOADPATH						
applName	REAL						
8	Connect the <b>buttonClickEvent</b> event from the <i>OK</i> push button to the <b>appendText</b> action of the IVBStringPart* part, <b>2</b> . This connection appends the upload path, given as a parameter, to the command file string.						
9	Connect the <b>actionResult</b> attribute from connection <b>1</b> to the <b>appendText</b> attribute of connection <b>2</b> (see connection <b>3</b> ). The upload directory is given as a parameter to the connection to be appended to the upload command.						
10	Connect the <b>buttonClickEvent</b> event from the <i>OK</i> push button to the <b>system</b> action from the systemCommand part <b>4</b> . This connection triggers the execution of the command file.						

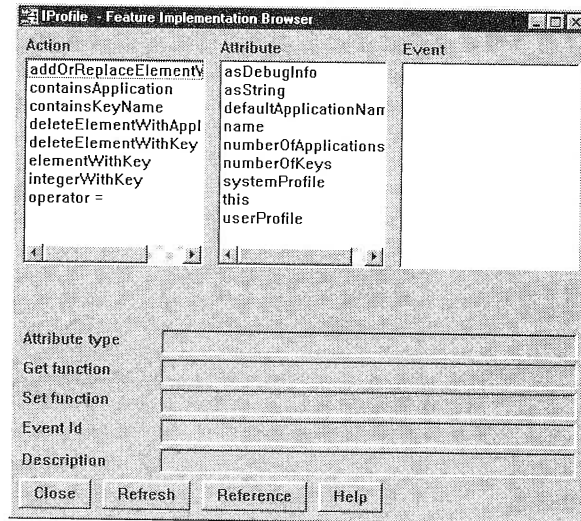
**Table 57.** (Part 3 of 3) Building AUploadView Part

Step	Action
11	Connect the <b>string</b> attribute from the variable to the <b>text</b> attribute from the connection <b>4</b> (see connection <b>5</b> ). The whole command line is passed to the connection as a parameter for the system command.
12	Connect the <b>buttonClickEvent</b> event from the <i>OK</i> push button to the <b>close</b> action of the frame window, <b>6</b> .
13	Connect the <b>buttonClickEvent</b> event from the <i>Cancel</i> push button to the <b>close</b> action of the frame window, <b>7</b> .
<b>Note:</b> The order of connections <b>1</b> , <b>2</b> , <b>4</b> , and <b>6</b> is crucial! Reverse highlighted letters and numbers are keyed to Figure 127 on page 325.	

You can now save your part and generate its code. From the Visual Builder window, select **Save and generate** → **Part source** in the **File** pull-down menu.

Building this part gives us the opportunity to introduce the `stdlibSamples` part from `VBSAMPLE.VBB`. This part provides you with many actions that correspond to the functions of the standard C library. You can think of this part as a wrapper for the standard C library. To use the part, just drop it on the free-form surface and connect an event of your application to one of its actions. You may have to draw some extra connections to provide the action with the necessary parameters. If you look in `VBSAMPLE.VBB`, you will find other useful parts, such as `stdioSamples` or `mathSamples`. The `stdioSamples` part is a wrapper for the standard C input/output library; the `mathSamples` part provides you with a convenient way of calling general-purpose mathematical functions.

You can do your “shopping” in this file, selecting the part that suits your needs. You can browse the features of a part by using the **Browse Part Features** option from the part’s pop-up menu (see Figure 128).



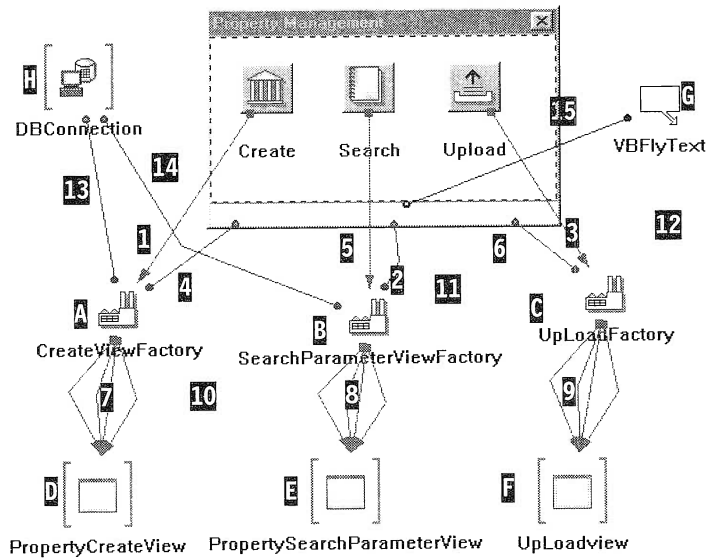
**Figure 128.** Browsing the IProfile Part's Features

You can, for example, check whether a part has some specific attributes that might be relevant in the context of your application and whether these attributes can be updated by some set member functions.

You cannot modify a part's feature from the browser. To modify a part's feature, use the Part Interface Editor.

## APropertyManagementView

APropertyManagementView is the primary view of the Property subsystem. From this view, the user can access the main functions of the subsystem: create a property, search a property, and generate export files. Each option is associated with a graphic push button that triggers the creation of an instance of APropertyCreateView, APropertySearchResultView, or AUploadView according to the option selected. You use three factory parts to build this view. The visual part instances created by each factory part are shown modally to the user (Figure 129).



**Figure 129.** APropertyManagementView

To build APropertyManagementView, follow the instructions in Table 58.

Table 58. (Part 1 of 2) Building APropertyManagementView Part	
Step	Action
1	Open the APropertyManagementView part.
2	Add three Factory* parts to the free-form surface, <b>A</b> , <b>B</b> , <b>C</b> , and change their respective type to APropertyCreateView*, APropertySearchParameterView*, and AUploadView*.
3	Add three IVBVariablePart* parts to the free-form surface, <b>D</b> , <b>E</b> , <b>F</b> , and change their respective type to APropertyCreateView*, APropertySearchParameterView*, and AUploadView*. Double click on <b>F</b> and initialize <i>generateCommandText</i> with DB2CMD PROPERTY.BAT (add one trailing blank to prevent concatenating the upload path parameter when the <b>appendText</b> action is called in AUploadView).
4	Add an IVBFlyText* part to the free-form surface, <b>G</b> .

**Table 58.** (Part 2 of 2) Building APropertyManagementView Part

Step	Action
5	Add an IDatastore* variable part to the free-form surface, <b>1</b> . This variable part represents the database connection established when the application starts. As explained in “Managing the Database Connection” on page 269, this database connection must be propagated, through variable parts, to the view that requires it.
6	Connect the <b>buttonClickEvent</b> event of the <i>Create</i> graphic push button to the <b>new</b> action of CreateViewFactory, <b>1</b> .
7	Connect the <b>buttonClickEvent</b> event of the <i>Search</i> graphic push button to the <b>new</b> action of SearchParameterViewFactory, <b>2</b> .
8	Connect the <b>buttonClickEvent</b> event of the <i>Upload</i> graphic push button to the <b>new</b> action of UpLoadViewFactory, <b>3</b> .
9	Connect the <b>this</b> attribute of the IFrameWindow* part to the <b>owner</b> attribute of each factory part, <b>4</b> , <b>5</b> , <b>6</b> .
10	<p>Connect each factory part to its corresponding variable with the following connections in the proper order, <b>7</b>, <b>8</b>, <b>9</b>:</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> newEvent → this</li> <li><input type="checkbox"/> newEvent → setFocus</li> <li><input type="checkbox"/> newEvent → showModally</li> <li><input type="checkbox"/> newEvent → deleteTarget</li> </ul> <p>Notice that the <b>deleteTarget</b> action is necessary to clean the memory because the view instances are shown modally.</p>
11	Connect the <b>this</b> of the DBConnection variable to the <b>dbConnection</b> attribute of CreateViewFactory, <b>10</b> . This attribute is an IDatastore* variable that is promoted in APropertyCreateView. With this connection, the database connection is transmitted to APropertyCreateView.
12	Connect the <b>this</b> attribute of the DBConnection variable to the <b>dbConnection</b> attribute of SearchParameterViewFactory, <b>11</b> . This attribute is an IDatastore* variable that is promoted in APropertySearchParameterView. With this connection, the database connection is transmitted to APropertySearchParameterView.
13	Connect the <b>this</b> attribute of the IInfoArea* part to the <b>longTextControl</b> attribute of the IVBFlyText* part, <b>12</b> . The long fly-over help texts are displayed in the info area. You can add your own long and short fly-over help texts to the controls of your choice.
<b>Note:</b> Reverse highlighted letters and numbers are keyed to Figure 129.	

In Step 3, the ***generateCommandText*** attribute is initialized with the command: `DB2CMD PROPERTY.BAT`. The DB2CMD prefixes the PROPERTY.BAT batch file to make sure the batch file is executed within the DB2 command line environment. A trailing space is added to the command to ensure that a space exists after the concatenation of the command line and the upload path parameter (See Steps 8 and 9 in Table 57 on page 325).

You can now save your part and generate its code. From the Visual Builder window, select **Save and generate** → **Part source** in the **File** pull-down menu.

## ALogonView

When the application starts, ALogonView enables the user to connect to the database. As with ADeleteDialogView, this view is so simple that we do not provide you with step-by-step instructions. In fact, there is only one connection to draw to complete the view:

1. Open ALogonView.
2. Connect the **buttonClickEvent** event of the ***Cancel*** push button to the **close** action of the frame window.

You can now save your part and generate its code. From the Visual Builder window, select **Save and generate** → **Part source** in the **File** pull-down menu.

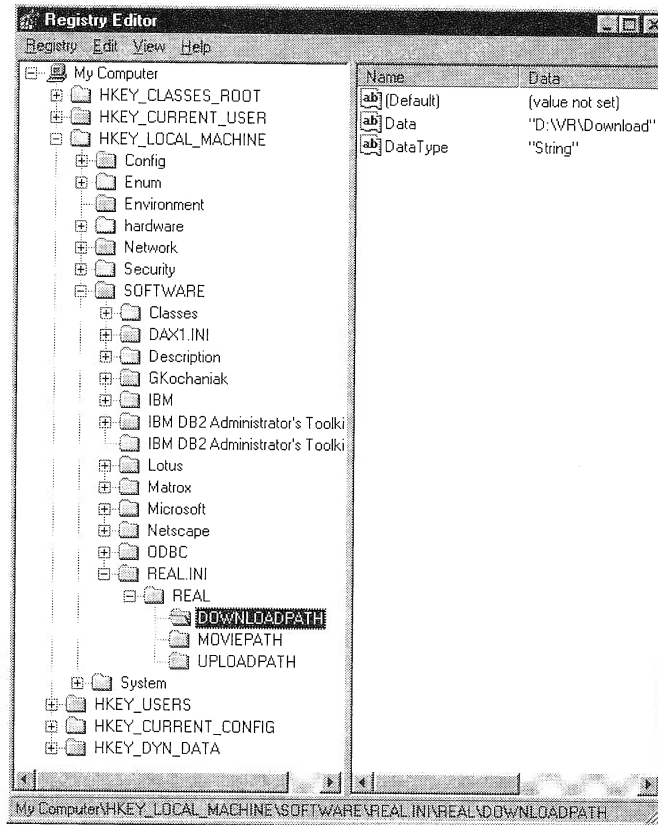
## ARealSettingsView

Windows NT and Windows 95 provide the programmer with a set of functions to organize, query, read, and write pieces of data in a centralized repository called the *registry*. The registry is a database of information that Windows programs, hardware, and administrators use to identify themselves to the system. They use this database to retrieve information on startup or during the execution of a program. Applications can use the registry to store specific information, such as the window position or font choice. The system itself uses the registry to store system configuration information. To make a distinction between all these kinds of information, the registry is broken up into major and minor keys. Each key can contain data items called value entries or they can contain additional subkeys. You can compare the registry structure to a DOS file system with directories and files.

The registry structure is simple (Figure 130). Each piece of data is identified by a key. Collections of data are combined into groups identified by an application name key. The application name key must be



unique in the registry. The subkeys must be unique within a given application key. To access data within the registry, you specify its application name key and its subkeys.



**Figure 130.** Structure of the Registry

The data specific to your application are stored under two specific keys:

- HKEY\_LOCAL\_MACHINE\SOFTWARE
- HKEY\_CURRENT\_USER\Software

The first key is used to store application-specific data, whereas the second key is used to store user-specific data.

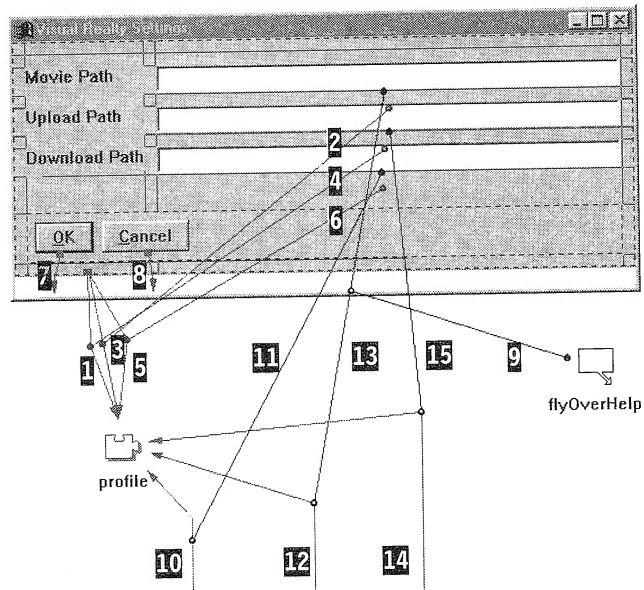
ARealSettingsView is responsible for maintaining the settings of our application in the registry. You store information about the different directories that are accessed during database upload and download under the key HKEY\_LOCAL\_MACHINE\SOFTWARE (see Figure 130 and “ARealSettingsView” on page 218):

- ☐ Movie path
- ☐ Upload path
- ☐ Download path

The information is stored in the `REAL.INI` subkey under the application name key `REAL`. The following keys are associated with the directories:

- ☐ `MOVIEPATH`, for the movie directory
- ☐ `UPLOADPATH`, for the upload directory
- ☐ `DOWNLOADPATH`, for the download directory

To maintain our application data, `ARealSettings` uses the services of an `IProfile` part (see Figure 131). `IProfile` part, like `IDate` part, is a class interface part that has no notification ability (see “Connecting a Nonvisual Part to a Visual Part” on page 249).



**Figure 131.** ARealSettings Part

Because `ARealSettingsView` is simple, we provide you with only one instruction table. To build this part, add the necessary subparts and make the connections as indicated in Table 59.

Table 59. (Part 1 of 3) Building ARealSettingsView Part	
Step	Action
1	Open the <code>ARealSettingsView</code> part.

**Table 59.** (Part 2 of 3) Building ARealSettingsView Part

Step	Action						
2	Add an IVBFlytext* part on the free-form surface, <b>A</b> .						
3	Add short and long fly-over help texts to the entry fields and the push buttons.						
4	<p>Add an IProfile* part on the free-form surface, <b>B</b>. Open its settings and set up the part as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>defaultApplicationName</td><td>REAL</td></tr> <tr> <td>name</td><td>REAL.INI</td></tr> </table> <p>REAL.INI holds the REAL application settings.</p>	Field	Value	defaultApplicationName	REAL	name	REAL.INI
Field	Value						
defaultApplicationName	REAL						
name	REAL.INI						
5	<p>Connect the <b>buttonClickEvent</b> event of the <i>OK</i> push button to the <b>addOrReplaceElementWithKey</b> action of the IProfile part <b>1</b>. Then, set the connection parameters as follows:</p> <table> <tr> <th>Field</th><th>Value</th></tr> <tr> <td>key</td><td>MOVIEPATH</td></tr> <tr> <td>applName</td><td>REAL</td></tr> </table> <p>The data parameter is retrieved from the corresponding entry field.</p>	Field	Value	key	MOVIEPATH	applName	REAL
Field	Value						
key	MOVIEPATH						
applName	REAL						
6	Connect the <b>text</b> attribute of the movie path entry field to the <b>data</b> attribute of the connection <b>1</b> (see <b>2</b> ).						
7	<p>Connect the <b>buttonClickEvent</b> event of the <i>OK</i> push button to the <b>addOrReplaceElementWithKey</b> action of the IProfile* part <b>3</b>. Then, set the connection parameters as follows:</p> <table> <tr> <th>Parameter</th><th>Value</th></tr> <tr> <td>key</td><td>UPLOADPATH</td></tr> <tr> <td>applName</td><td>REAL</td></tr> </table> <p>The data parameter is retrieved from the corresponding entry field.</p>	Parameter	Value	key	UPLOADPATH	applName	REAL
Parameter	Value						
key	UPLOADPATH						
applName	REAL						
8	Connect the <b>text</b> attribute of the upload path entry field to the <b>data</b> attribute of the connection <b>3</b> (see <b>4</b> ).						
9	<p>Connect the <b>buttonClickEvent</b> event of the <i>OK</i> push button to the <b>addOrReplaceElementWithKey</b> action of the IProfile* part <b>5</b>. Then, set the connection parameters as follows:</p> <table> <tr> <th>Parameter</th><th>Value</th></tr> <tr> <td>key</td><td>DOWNLOADPATH</td></tr> <tr> <td>applName</td><td>REAL</td></tr> </table> <p>The data parameter is retrieved from the corresponding entry field.</p>	Parameter	Value	key	DOWNLOADPATH	applName	REAL
Parameter	Value						
key	DOWNLOADPATH						
applName	REAL						
10	Connect the <b>text</b> attribute of the download path entry field to the <b>data</b> attribute of the connection <b>5</b> (see <b>6</b> ).						

**Table 59.** (Part 3 of 3) Building ARRealSettingsView Part

Step	Action						
11	Connect the <b>buttonClickEvent</b> event from the <i>OK</i> push button to the <b>close</b> action of the frame window, <b>7</b> .						
12	Connect the <b>buttonClickEvent</b> event from the <i>Cancel</i> push button to the <b>close</b> action of the frame window, <b>8</b> .						
13	Connect the <b>this</b> attribute from the window info area to the <b>long-TextControl</b> attribute of the fly-over help, <b>9</b> .						
14	<p>Connect the <b>ready</b> event from the free-form surface to the <b>elementWithKey</b> action of the IProfile* part, <b>10</b>. Then, set the connection parameters as follows:</p> <table> <tr> <th>Parameter</th><th>Value</th></tr> <tr> <td>key</td><td>DOWNLOADPATH</td></tr> <tr> <td>applName</td><td>REAL</td></tr> </table> <p>The data parameter is retrieved from the profile at startup.</p>	Parameter	Value	key	DOWNLOADPATH	applName	REAL
Parameter	Value						
key	DOWNLOADPATH						
applName	REAL						
15	Connect the <b>actionResult</b> attribute of connection <b>10</b> to the <b>text</b> attribute of the download path entry field, <b>11</b> .						
16	<p>Connect the <b>ready</b> event from the free-form surface to the <b>elementWithKey</b> action of the IProfile* part, <b>12</b>. Then, set the connection parameters as follows:</p> <table> <tr> <th>Parameter</th><th>Value</th></tr> <tr> <td>key</td><td>UPLOADPATH</td></tr> <tr> <td>applName</td><td>REAL</td></tr> </table> <p>The data parameter is retrieved from the profile at startup.</p>	Parameter	Value	key	UPLOADPATH	applName	REAL
Parameter	Value						
key	UPLOADPATH						
applName	REAL						
17	Connect the <b>actionResult</b> attribute of connection <b>12</b> to the <b>text</b> attribute of the upload path entry field, <b>13</b> .						
18	<p>Connect the <b>ready</b> event from the free-form surface to the <b>elementWithKey</b> of the IProfile* part, <b>14</b>. Then, set the connection parameters as follows:</p> <table> <tr> <th>Parameter</th><th>Value</th></tr> <tr> <td>key</td><td>MOVIEPATH</td></tr> <tr> <td>applName</td><td>REAL</td></tr> </table> <p>The data parameter is retrieved from the profile at startup.</p>	Parameter	Value	key	MOVIEPATH	applName	REAL
Parameter	Value						
key	MOVIEPATH						
applName	REAL						
19	Connect the <b>actionResult</b> attribute of connection <b>14</b> to the <b>text</b> attribute of the movie path entry field, <b>15</b> .						
<b>Note:</b> The order of connections <b>1</b> , <b>3</b> , <b>5</b> , and <b>7</b> is crucial! Reverse highlighted letters and numbers are keyed to Figure 131 on page 333.							

You can now save your part and generate its code. From the Visual Builder window, select **Save and generate** → **Part source** in the **File** pull-down menu.

## ARealMainView

When the application starts, ARealMainView is displayed. From this view, the user can log on to the database, change the application settings, or access one subsystem. To access one of the subsystems, the user must establish a connection to the database. In the sections that follow, you build the ARealMainView part in four steps:

1. Implement the logon function that enables the user to connect to the database.
2. Build the access to the settings of the application and to the Property subsystem.
3. Tailor the fly-over help feature, using the attribute tear-off facility.
4. Add help to the application.

## DB2 for Windows Authentication

In DB2 for Windows, different type of authentications are provided which determine where authentication occurs:

**SERVER** specifies that authentication occurs on the server. The user name and password specified during the connection or attachment attempt are compared to the valid user name and password combinations on the server to determine if the user is permitted to access the database.

**CLIENT** specifies that authentication occurs on the node where the application is invoked. The user name and password specified during a connection or attachment attempt are compared with the valid user name and password combinations on the client to determine if the user name is permitted access to the database. No further authentication will take place on the database server.

**DCS** specifies how authentication will take place for databases accessed using Distributed Database Connection Services for Windows NT (DDCS).

Basically two access scenarios are possible: local access or remote access.

---

### ***Local and Domain Database Access***

Windows NT always requires a user to log on to a workstation or domain before granting access to the Desktop. Once the user has been authenticated by the operating system, the user may attempt to perform database activities.

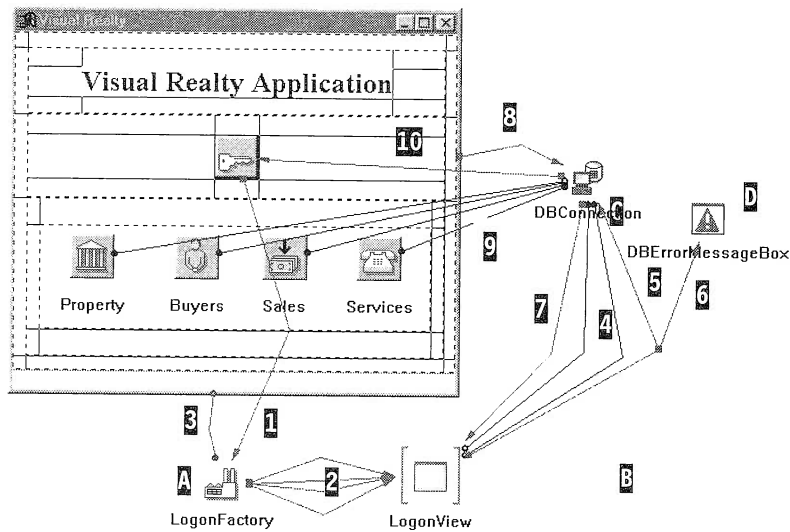
If the user performs a workstation logon, the user knows only that workstation. When DB2 user attempts to access a remote database that has SERVER authentication, the user will be required to supply a user name and password. The DB2 for Windows NT server code detects whether a connection is local or remote. For local connections, when authentication is SERVER, additional verification of the user is not performed. In this case, ALogonView can be bypassed by clicking the **OK** push-button without providing user name and password. For remote connections, the user is validated using the specified user name and password and ALogonView transmits the authentication information to the server.

### ***Remote Database Access***

If the remote instance has CLIENT authentication, a user name and password are not required. If the remote instance has SERVER authentication, the user must provide a user name and password although the user has already logged on to the local machine or to the domain. In this specific scenario, ALogonView provides a way of getting the authentication information and passing it to the server.

## **Logging on to the Database**

The **logon** push button enables the user to access the logon view. From this view, the user can enter a user ID and a password to get connected to the REAL database. ARealMainView uses the ALogonView services to collect the logon information (Figure 132). ALogonView, in turn, uses the IDatastore services to connect to the database. If an exception occurs during the logon procedure (wrong user ID or password), a message box warns the user and prompts for the user ID and password to be reentered.



**Figure 132.** Logon to the Database

To implement the logon function, follow the steps in Table 60.

**Table 60.** Adding Parts for the Logon Function

Step	Action
1	Open the ARealMainView part.
2	Add an IVBFactory* part, <b>A</b> , to the free-form surface and change its type to ALogonView*. This part creates the instance of ALogonView to get the user ID and password.
3	Add an IVBVariablePart* part, <b>B</b> , to the free-form surface and change its type to ALogonView*. This part represents the ALogon-View instance created by the factory.
4	Add an IDatastore* part to the free-form surface, <b>C</b> . Open its settings and fill in the <i>datastoreName</i> attribute with the name of the database: <b>REAL</b> . This part which represents the connection to the database, is transmitted to each subsystem to allow each to access the different tables (see “Managing the Database Connection” on page 269).
5	Add an IMessageBox* part, <b>D</b> , to the free-form surface. This message box displays a warning message if the connection fails (because of incorrect authentication, for example).

**Note:** Reverse highlighted letters are keyed to Figure 132 on page 338.

Once you have placed the parts on the free-form surface, make the connections as shown in Table 61.

<b>Table 61.</b> (Part 1 of 2) Connecting Parts for the Logon Function		
<b>Key</b>	<b>Connection</b>	<b>Description</b>
<b>1</b>	buttonClickEvent → new	Create an instance of ALogonView to get the user authentication.
<b>Note:</b> The order of the next four connections is crucial!		
<b>2</b>	<input type="checkbox"/> newEvent → this <input type="checkbox"/> newEvent → setFocus <input type="checkbox"/> newEvent → showModally <input type="checkbox"/> newEvent → deleteTarget	Associate the ALogonView variable with the instance created by the factory. ALogonView is displayed as a modal window.
<b>3</b>	owner → this	The frame window is the owner of the instance created by the factory (ALogonView is shown modally).
<b>4</b>	<input type="checkbox"/> EntryFieldUserIDText → userName <input type="checkbox"/> EntryFieldPasswordText → authentication	Transmit the user ID and the password to IDatastore.
<b>5</b>	PushButtonOKClickEvent → connect	Connect to the database when the user validates his or her user ID and password.
<b>6</b>	exceptionOccurred → showException	The exception is returned by the message box, and the user can try to log on again.
<b>7</b>	connected → close	If the connection is established, the logon window is closed.
<b>8</b>	close → disconnect	Close the connection when the main window is closed.
<b>9</b>	<input type="checkbox"/> isConnected → enabled (Properties graphic push button) <input type="checkbox"/> isConnected → enabled (Buyers graphic push button) <input type="checkbox"/> isConnected → enabled (Sales graphic push button) <input type="checkbox"/> isConnected → enabled (Services graphic push button)	Enable the different graphic push buttons to let the user access the subsystems when the database connection is established. Notice that the graphic push buttons are initially disabled at startup (see “ARealMainView” on page 220).



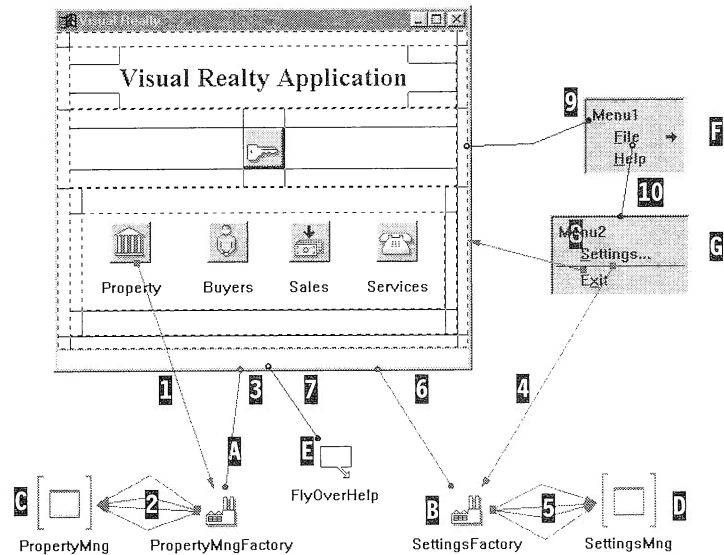
**Table 61.** (Part 2 of 2) Connecting Parts for the Logon Function

Key	Connection	Description
<b>10</b>	isConnected → disable	Disable the logon graphic push button to prevent the user from connecting again to the database when the connection is established.
<b>Note:</b> Reverse highlighted numbers are keyed to Figure 132 on page 338. To keep the drawings simple, we do not key all connections, and we do not show ARealMainView with its multicell canvases.		

Notice that connection **10** is fired whenever the connection status of the IDatastore\* part changes. Here you do not have to use another Boolean part to ensure that the value is changed to TRUE. In effect, at startup, this value is set to FALSE. Thus, you can be sure that the first time it changes, it is set to TRUE.

## Accessing the Application Settings and the Property Subsystem

Once the database connection is established, the user can access the Property subsystem. From the menu bar, the user can also access the settings of the application and provide the directories required, to upload or download the database to or from the server. In both cases, you use a factory to dynamically create an instance of the corresponding view (see Figure 133).



**Figure 133.** Application Settings and Property Subsystem Access

To complete the ARealMainView part, follow the steps in Table 62.

**Table 62.** (Part 1 of 2) Adding Parts to Access the Application Settings and Property Subsystem

Step	Action
1	Add two IVBFactory* parts, <b>A</b> and <b>B</b> , to the free-form surface and change their respective types to APropertyManagementView* and ARealSettingsView*. These parts create the instances of APropertyManagementView and ARealSettingsView.
2	Add two IVBVariablePart* parts, <b>C</b> and <b>D</b> , to the free-form surface and change their respective types to APropertyManagementView* and ARealSettingsView*. These parts represent the APropertyManagementView and ARealSettingsView instances created by each factory.
3	Add an IVBFlyText* part to the free-form surface, <b>E</b> . This part enables the fly-over help to be displayed. The short help text is displayed as a bubble help and the long help text is displayed in the window info area (see “Adding Fly-over Help to a Control” on page 270).
4	Add an IMenu* part, <b>F</b> , to the free-form surface.

**Table 62.** (Part 2 of 2) Adding Parts to Access the Application Settings and Property Subsystem

Step	Action
5	Add an IMenu* part to the first menu and change its label to <b>~File</b> (this is a shortcut to build a cascade menu). The cascade menu, <b>G</b> , enables the user to access the application settings or exit the application.
6	Add an IMenuItem* part to the first menu part and change its label to <b>~Help</b> . Open the menu item settings notebook and select <b>help-Command</b> in the <i>Command type</i> list box. From this option, the user accesses the application general help.
7	Add an IMenuItem* part to the second menu part and change its label to <b>~Settings...</b> . Open the notebook settings of the menu item and add <b>ALT+S</b> as the accelerator key: Check the <b>ALT</b> check box in the <i>Accelerator</i> group box and select the <b>S</b> key in the key drop-down list. From this option, the user accesses the application settings.
8	Add an IMenuSeparator* part to the second menu part.
9	Add an IMenuItem* part to the second menu part and change its label to <b>E~xit</b> . Open the notebook settings of the menu item and add <b>F3</b> as the accelerator key: Select the <b>F3</b> key in the key drop-down list. From this option, the user can exit the application.
10	Add short and long fly-over texts to every graphical push button and menu item.
<b>Note:</b> Reverse highlighted letters are keyed to Figure 133 on page 341.	

Once you have placed the parts on the free-form surface, make the connections as shown in Table 63.

<b>Table 63.</b> (Part 1 of 2) Connecting Parts for Property Subsystem and Settings Access		
<b>Key</b>	<b>Connection</b>	<b>Description</b>
<b>1</b>	buttonClickEvent → new	Create an instance of APropertyManagementView.
<b>Note:</b> The order of the connections from the factory is crucial (see “Using an Object Factory to Update the Database” on page 295).		
<b>2</b>	<input type="checkbox"/> newEvent → this <input type="checkbox"/> newEvent → setFocus <input type="checkbox"/> newEvent → showModally <input type="checkbox"/> newEvent → deleteTarget	Associate the AProperty-ManagementView variable part with the instance created by the factory. APropertyManagementView is displayed as a modal window.
<b>3</b>	owner → this	The frame window is the owner of the instance created by the factory (APropertyManagementView is shown modally).
<b>4</b>	commandEvent → new	Create an instance of ARealSettingsView.
<b>Note:</b> The order of the connections from the factory is crucial (see “Using an Object Factory to Update the Database” on page 295).		
<b>5</b>	<input type="checkbox"/> newEvent → this <input type="checkbox"/> newEvent → setFocus <input type="checkbox"/> newEvent → showModally <input type="checkbox"/> newEvent → deleteTarget	Associate the ARealSettingsView variable part with the instance created by the factory. ARealSettingsView is displayed as a modal window.
<b>6</b>	owner → this	The frame window is the owner of the instance created by the factory (ARealSettingsView is shown modally).
<b>7</b>	this → longTextControl	Display the long fly-over help in the window info area.
<b>8</b>	commandEvent → close	Close the window when this option is selected.
<b>9</b>	menu → this	The menu is associated with the main window.

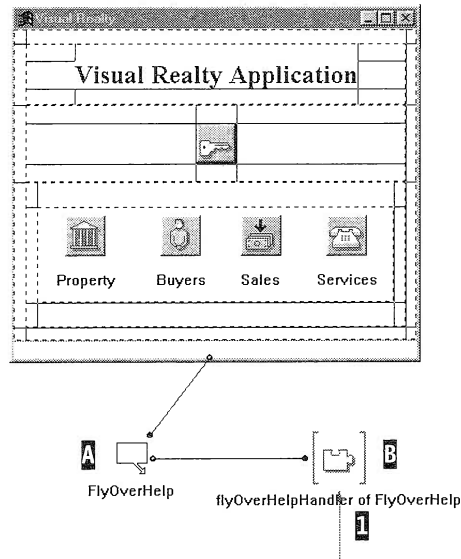
Table 63. (Part 2 of 2) Connecting Parts for Property Subsystem and Settings Access		
Key	Connection	Description
10	menu → this	The submenu is associated with the first menu item. This connection is built automatically when you drop a menu part on another menu part.
<b>Note:</b> Reverse highlighted numbers are keyed to Figure 133 on page 341. To keep the drawings simple, we do not key all connections, and we do not show ARealMainView with its multicell canvases.		

To transmit the database connection to APropertyManagementView, you must connect the *this* attribute of the IDatastore part, DBConnection, to the *dbConnection* promoted variable of APropertyManagementView (this connection is not shown on Figure 133 to avoid overloading the figure).

### Tearing Off an Attribute

Fly-over help is a great feature for a novice user, but it can become annoying when the user gains experience. Having a bubble help appear each time you position the mouse pointer on a control to tell you what you already know is irritating. To tailor the fly-over help behavior, you can set up the delay between the time the pointer has stopped on a control and the time the bubble help is displayed. To do so, access the event handler of the fly-over help part and set its *delayTime* attribute to the value of your choice (the value is given in milliseconds).

It is not possible to directly access the *delayTime* attribute from an IVBFlyText\* part. Rather, you can tear off its *flyOverHelpHandler* attribute and from it access the *delayTime* attribute. The torn-off attribute is not a separate part, but a variable that represents the attribute itself (Figure 134). Tearing off an attribute is like peeling an onion: From a part, you can tear off one of its attributes; then from that attribute, you can tear off another attribute, and so on.



**Figure 134.** Tearing Off an Attribute

To tailor the fly-over help behavior of ARealMainView, follow these six steps:

1. Select the viewText\* part on the free-form surface, **A**.
2. Select the **Tear-Off Attribute** option from its pop-up menu. A list of the IVBFlyText\* part's attributes is displayed.
3. Select **flyOverHelpHandler** from the list. The torn-off attribute, **B**, is created on the free-form surface as a variable that points to the **flyOverHelpHandler** attribute of the IVBFlyText\* part.
4. Connect the **ready** event from the free-form surface to the **delay-Time** action of the torn-off attribute, **1**.
5. Double-click on the connection and click on the **Set parameters...** push button to set the delay time.
6. Type in the delay time in milliseconds. One thousand milliseconds (one second) is a good value.

**Tip**

Tearing off an attribute is quite different from promoting an attribute. When you tear off an attribute, you create a reference that points to the attribute and an attribute-to-attribute connection that associates the attribute with the reference. Because most of the time the attribute is a part itself (as opposed to a primitive type such as integer or real), you can access its attributes and methods. When you promote an attribute, you generate an extra member function, which enables a composite part to access its promoted attribute. When you use the tear-off facility, you must not forget that each torn-off attribute involves creation of an attribute-to-attribute connection and its associated class. This requirement can introduce excessive overhead in your application if the torn-off attribute is not justified.

## Adding Help to Your Application

You can provide your application with two basic help types:

**General help**

Provides general information for a specific window, explaining its purpose and how it operates.

**Context-sensitive help**

Provides help information for the current choice, object, or group of choices or objects. The user can display the context-sensitive help by tabbing or moving the cursor to an object and either:

- Pressing the **F1** key (this is automatically handled by Windows)

or

- Selecting the **Help** push button or menu option if you have provided one

In our application we used both types of help.

Before connecting any help files to the application, you must first create them. For convenience we have provided two help sources:

- ☐ MAIN.IPF, the general help
- ☐ SETTINGS.IPF, the application settings help

These IPF files are compiled during the make phase of your Help project, that you set up in “Creating the Help Project” on page 116.

If you prefer, you can use the IPF compiler directly (the IPF compiler comes with VisualAge for C++ for Windows) to generate a binary help file (these files have an HLP extension) for each source file by issuing the `ipfc IPF_file` command.

Then, place your HLP files in the `D:\VR\HELP` directory to make them accessible to your application. Do not forget to add the `D:\VR\HELP` directory in the `HELP` variable of your system variables.

#### Portability



If you want your help to be usable in both OS/2 and Windows, write your help source code in Information Presentation Facility (IPF) format. If portability is not a concern for your Windows application, you can write your help source code in Rich Text Format (RTF). For information about creating help source in RTF, see your Windows documentation.

To connect the help to the application, add an `IHelpWindow*` part on the free-form surface and set its ***Help libraries*** attribute to the associated help file name (this attribute is accessible from the settings notebook of the `IHelpWindow*` part).

To provide general help to your application, open the settings notebook of your main window and, on the Control page, enter the resource number for the general information help panel in the *Help panel id* entry field.

To provide a context-sensitive help to a subpart, open the subpart settings notebook and, on the Control page, enter the resource number for the specific information help panel in the *Help panel id* entry field.

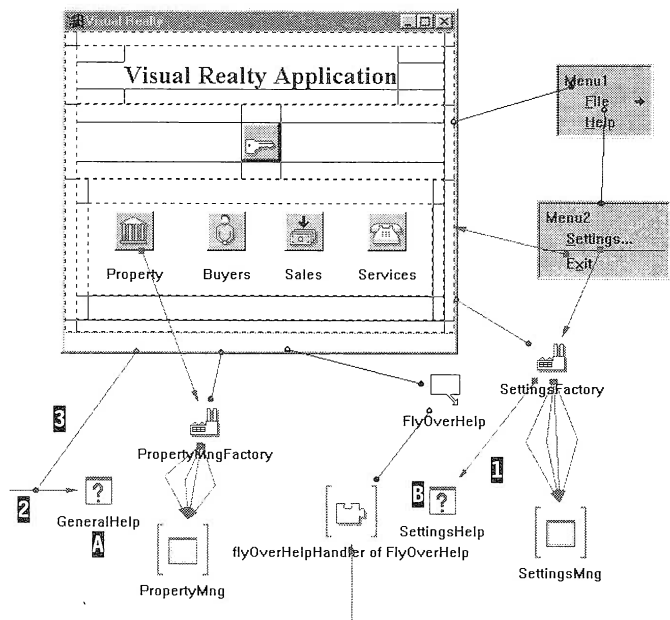
You can associate a different help file with each application subsystem in two ways:

- ❑ You can add an `IHelpWindow*` part to the free-form surface where the main window subsystem is located and set the ***Help libraries*** attribute to the corresponding help file. In this way, you statically associate the help window with the main window subsystem. Use this solution when you have created multiple help library files for your application. You will use this method to connect the `MAIN.HLP` file to our main window (see the GeneralHelp part in Figure 135).
- ❑ You can dynamically associate a help window with a window that is generated by an object factory. In this case, you add an `IHelpWindow*` part to the free-form surface of the view that holds the factory and set the ***Help libraries*** attribute to the help file of the child window. Then, connect the ***newEvent*** event from the object factory to the ***setAssociatedWindow*** action of the help window.



You will use this method to connect the SETTINGS.HLP help file to ARealSettings (see the SettingsHelp part in Figure 135). This solution can be used when you have one help library file for all of the panels of one subapplication. This solution enables you to gather all of the IHelpWindow\* parts in the same view, thus facilitating the maintenance of several help library files.

If you have several IHelpWindow\* parts in one view, you must use the **setAssociatedWindow** action explicitly to associate a window with its corresponding help library file.



**Figure 135.** Adding Help to the Application

To add a help facility to the application, follow the instructions in Table 64.

Table 64. (Part 1 of 2) Adding Help to the Application	
Step	Action
1	Add two IHelpWindow* parts, A and B, to the free-form surface. (The IHelpWindow* part is located in the <i>Other</i> category.)

**Table 64.** (Part 2 of 2) Adding Help to the Application

Step	Action						
2	<p>Open the settings window of each IHelpWindow* part and fill in the <i>Help libraries</i> entry fields as follows:</p> <table> <tr> <th>Part</th><th>Help libraries</th></tr> <tr> <td>GeneralHelp</td><td>REAL.HLP</td></tr> <tr> <td>SettingsHelp</td><td>SETTINGS.HLP</td></tr> </table> <p>The corresponding help files are associated with their corresponding IHelpWindow* part.</p>	Part	Help libraries	GeneralHelp	REAL.HLP	SettingsHelp	SETTINGS.HLP
Part	Help libraries						
GeneralHelp	REAL.HLP						
SettingsHelp	SETTINGS.HLP						
3	Connect the <b>newEvent</b> event of ARealSettings factory to the <b>setAssociatedWindow</b> action of SettingsHelp, <a href="#">1</a> .						
4	Connect the <b>ready</b> event of ARealMainView to the <b>setAssociatedWindow</b> action of GeneralHelp, <a href="#">2</a> .						
5	Connect the <b>this</b> attribute of the IFrameWindow* to the <b>associatedWindow</b> parameter of connection <a href="#">2</a> , (see <a href="#">3</a> ).						

You must now add the resource numbers to the settings of ARealMainView (Table 65) and ARealSettingsView (Table 66) to access the specific help information.

**Table 65.** (Part 1 of 2) Adding Help Resource Numbers for ARealMainView

Step	Action												
1	Open the settings notebook of the IFrameWindow* part.												
2	Switch to the Control page and enter <b>100</b> in the <i>Help panel id</i> entry field.												
3	<p>Open the notebook settings of the graphical push button and, on the Control page, set the <i>Help panel id</i> as follows:</p> <table> <tr> <th>Button</th><th>Help panel id</th></tr> <tr> <td>Logon</td><td>110</td></tr> <tr> <td>Properties</td><td>120</td></tr> <tr> <td>Buyers</td><td>130</td></tr> <tr> <td>Sales</td><td>140</td></tr> <tr> <td>Services</td><td>150</td></tr> </table>	Button	Help panel id	Logon	110	Properties	120	Buyers	130	Sales	140	Services	150
Button	Help panel id												
Logon	110												
Properties	120												
Buyers	130												
Sales	140												
Services	150												

**Table 65.** (Part 2 of 2) Adding Help Resource Numbers for ARealMainView

Step	Action										
4	Open the notebook settings of the menu items and, on the Control page, set the <i>Help panel id</i> as follows: <table> <tr> <th>Menu item</th><th>Help panel id</th></tr> <tr> <td>File</td><td>160</td></tr> <tr> <td>Help</td><td>170</td></tr> <tr> <td>Settings</td><td>180</td></tr> <tr> <td>Exit</td><td>190</td></tr> </table>	Menu item	Help panel id	File	160	Help	170	Settings	180	Exit	190
Menu item	Help panel id										
File	160										
Help	170										
Settings	180										
Exit	190										
5	Save the part and regenerate its source code.										

**Table 66.** Adding Help Resource Numbers for ARealSettingsView

Step	Action								
1	Open the ARealSettingsView part.								
2	Open the settings notebook of the IFrameWindow* part.								
3	Switch to the Control page and enter <b>200</b> in the <i>Help panel id</i> entry field.								
4	Open the notebook settings of each entry field and, on the Control page, set the <i>Help panel id</i> as follows: <table> <tr> <th>Entry field</th><th>Help panel id</th></tr> <tr> <td>EntryFieldMovie</td><td>210</td></tr> <tr> <td>EntryFieldUpLoad</td><td>220</td></tr> <tr> <td>EntryFieldDownLoad</td><td>230</td></tr> </table>	Entry field	Help panel id	EntryFieldMovie	210	EntryFieldUpLoad	220	EntryFieldDownLoad	230
Entry field	Help panel id								
EntryFieldMovie	210								
EntryFieldUpLoad	220								
EntryFieldDownLoad	230								
5	Save the part and regenerate its source code.								

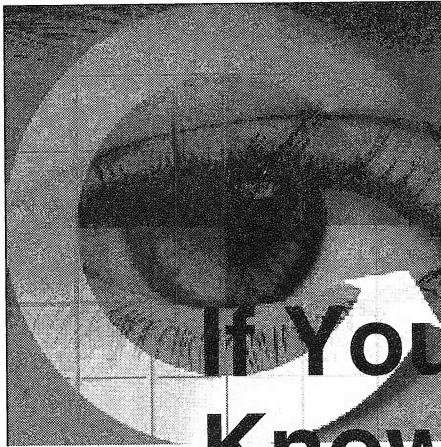
**Tip**

When you generate the code for your application, Visual Builder creates a help table in the resource file (.rc) that it generates. It inserts in the help table the resource number that you specify on the Control page of each subpart. From the resource file, you can translate your help file without recompiling the entire application.



Congratulations! You have built your first subsystem. Now we suggest that you compile your code and run it in the WorkFrame environment. You are now ready to take the plunge and explore other subsystems to discover new tricks for your future applications. In the next chapter, we go a step further and reveal the magic of Visual Builder: the notification framework.

# Part 4



## If You Want to Know More...

Now that you have developed the sample application, you must feel more confident with Visual Builder and Data Access Builder. In this part we cover some other important features of VisualAge for C++:

- ❑ The notification framework of the Visual Builder is explained in a comprehensive way which let you understand how the code is generated and how to take advantage of it in order to transform your C++ classes into Visual Builder nonvisual parts.
- ❑ More information is provided on Data Access Builder. We show you how to use ODBC as a data access method and how to implement the REAL database as simple text files.
- ❑ The SOM technology is explained by example, and we show you how to use Direct-to-SOM parts in the Visual Builder.
- ❑ Throughout several examples, you learn how to use the Compound Document Framework to make your applications OLE enabled.



# 10

## More about Visual Builder...

*Fortune favors the bold.*

-Virgil

This chapter takes you behind the scenes of Visual Builder. After reading this chapter, you will be able to understand how drawing a connection between two parts makes them cooperate and how the code is generated. You must understand those concepts if you want to import your existing C++ code as parts in Visual Builder or build your own parts.

### Notification Framework Concepts

The world of Visual Builder is ruled by notifiers and observers. A *notifier* enables other objects to register themselves as dependent on any change in the properties of the notifier. In other words, an object can tell a notifier: *I have a value that depends on one of your attributes. Could you notify me when the attribute value changes?*

Each notifier maintains a list of objects that are interested in a certain event. To register itself to a notifier, an object adds an *observer* to that list. Notifiers are responsible for publishing their supported notification events, managing the list of observers, and notifying observers when an event occurs.

Notifier objects indicate the notification events that they support by providing a series of unique identifiers in their interface. In Visual Builder, a notification ID is a unique string built from the class name and the attribute or event name, for example, `IPushButton::text`.

Whenever an event occurs, the notifier object sends out its notifications to all of its dependent observers. The observer can choose either to handle or ignore a notification by checking the event notification ID.

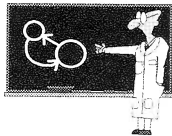
Observers and notifiers are defined in the IBM Open Class Library notification framework, respectively, as the *IObserver* and *INotifier* abstract classes.

## How Visual Builder Uses the Notification Framework

In Visual Builder, all parts are notifiers. Drawing a connection between two parts is equivalent to creating an observer for the *source* of the connection. The connection is then responsible for updating its *target* part whenever the source undergoes any state change.

Visual parts, such as `IEntryField`, generally inherit from the `IWindow` class, whereas nonvisual parts derive from the `IStandardNotifier` class. Both `IWindow` and `IStandardNotifier` provide a concrete implementation of the notifier protocol.

### Technical Information



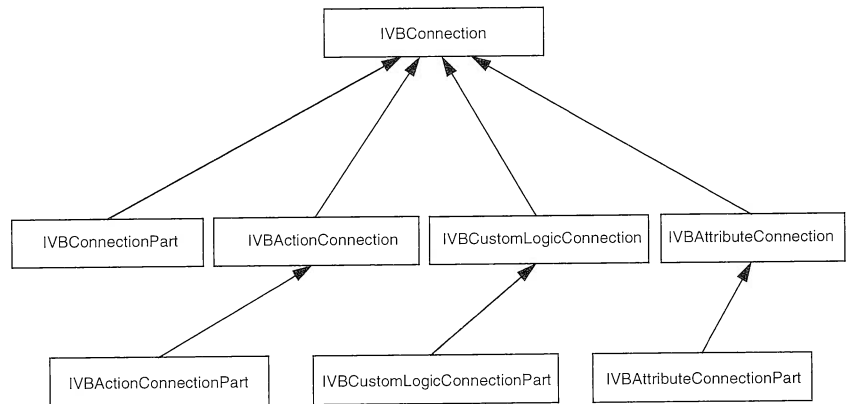
Throughout this chapter, we describe connections behavior in general. However, attribute-to-attribute connections are handled differently by Visual Builder. Therefore, we provide special information on attribute-to-attribute connections as needed in boxes like this one.

### Visual Builder Connections Classes Definition

Visual Builder defines several classes representing connections. As shown in Figure 136, all connections are derived from the `IVBConnection` base class. The `IVBConnection` class, which derives from the `IOb-`

server class, defines the basic behavior of a connection as an observer. If you examine the graph, you notice that classes are defined according to the connection type, such as `IVBAttributeConnection` or `IVBCustomLogicConnection`.

You also notice that, for each connection type, two classes are defined, for example `IVBConnection` and `IVBConnectionPart`. You need `IVBConnectionPart` and its counterparts each time a connection needs to become a notifier. See “Using Connections as Notifiers” on page 361 for more details on this topic.



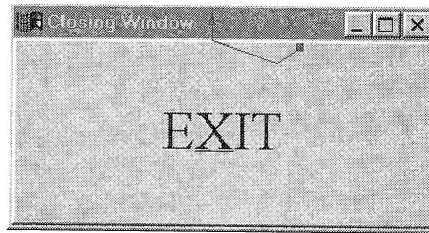
**Figure 136.** `IVBConnection` Class and its Derived Classes

### ***Scenario for a Connection***

Let us take a simple example. With the Composition Editor, create a frame window with a single client control: a push button. (Note that you must remove the default canvas that is created by Visual Builder if you want your sample to look like the one shown in Figure 137). Then, draw an event-to-action connection between the **buttonClick-Event** event of the push button and the **close** action of the frame window.



Then, rename the two parts `mainWindow` and `closePB`, respectively.



**Figure 137.** Sample Window: Using an Event-to-Action Connection

The scenario is the following: We want to understand what code is generated and the notification flow triggered when the user of our sample application clicks the **Exit** button.

### **Connection Class Definition**

For the connection, Visual Builder generates a child class of the `IVB-Connection` class as follows:

```
class closingWindowConn0 : public IVBConnection
{
public:
    //-----
    // Constructors / destructors
    //-----

    closingWindowConn0(IPushButton * aSource, IFrameWindow * aTarget) :
        source(aSource), target(aTarget)
    {...};

    virtual ~closingWindowConn0()
    {...};

    ...

private:
    //-----
    // private member data
    //-----

    IPushButton * source;
    IFrameWindow * target;

}; //closingWindowConn0
```

## Parts and Connection Initialization

First of all, the instances of our main actors are created:

```
iclosePB = new IPushButton (IC_FRAME_CLIENT_ID,...);
```

### Note

The mainWindow is not created now. In fact the frameWindow represents the part itself. An instance of mainWindow is created in the main application file (created when you choose **Generate main() for part...**). The mainWindow is therefore represented by the *this* keyword.

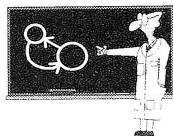
Each part maintains a list of the connections that are created. This list is used to initialize all connections, or delete all of them when the part is destroyed.

```
iclosingWindowConnectionList = new IVBConnectionList();
...
closingWindowConn0 *conn0= new closingWindowConn0(iclosePB, this);
iclosingWindowConnectionList->add (conn0);
```

A connection must register itself as an observer of the source part. This registration process is done in the connection constructor, and performed through the *handleNotificationFor()* method defined in the IObserver class:

```
closingWindowConn0(IPushButton * aSource, IFrameWindow * aTarget) :
    source(aSource), target(aTarget)
{
    handleNotificationsFor(*source);
};
```

### Technical Information



When using an attribute-to-attribute connection, you expect the values of the two attributes to always be synchronized. Thus, an attribute-to-attribute connection needs to register itself as an observer for both the source and the target of the connection. The constructor of an attribute-to-attribute connection, which derives from the IVBAttribute connection is defined as follows:

```
closingWindowConn1(IEntryField* aSource,
                  IStaticText* aTarget)
:source(aSource), target(aTarget)
{
    handleNotificationsFor(*source);
    handleNotificationsFor(*target);
}
```

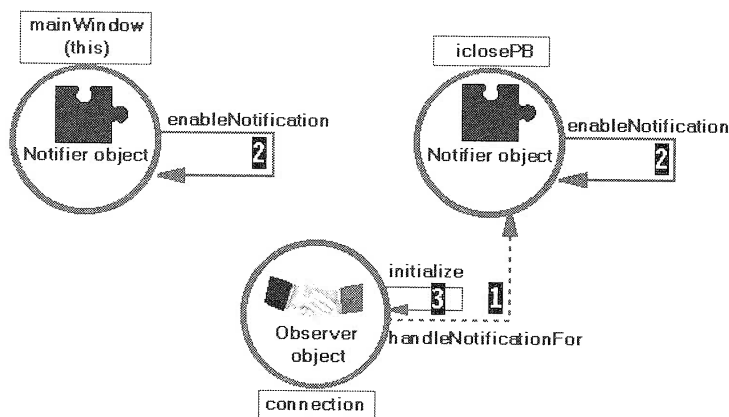
Notifiers are created disabled and must be enabled before they can notify any event. Therefore, Visual Builder parts can initialize themselves before the support for notification is effectively enabled. Notification enabling is provided by the *enableNotification()* method defined in the *INotifier* abstract class. For our example, the push button and frame window objects are enabled as follows:

```
this->enableNotification();
iclosePB->enableNotification();
```

## Reminder

Visual Builder always prefixes the name of the parts you create with an **i** (for example `iclosePB`).

The connection is now ready to handle any event data that a notifier sends. Figure 138 summarizes the initialization process.



**Figure 138.** Visual Builder Parts Initialization Process

### Notification Flow

Let us now click the **Exit** push button and try to understand the sequence of events in the system. The **buttonClickEvent** event has a unique notification ID, namely, *IButton::buttonClickId*. Each time you click a button, the *select()* method is called. (The *select()* method is implemented in the *ISettingButton* class, an ancestor of the *IPushButton* class.) It is the responsibility of the *select()* method to use the *notifyObservers()* method to indicate that the button has been clicked.

```

ISettingButton& ISettingButton::select(Boolean sel)
{
    ...
    notifyObservers(INotificationEvent(IButton::buttonClickId, *this, false));
    ...
    return *this;
}

```

Whenever the `notifyObservers()` method is called, each observer applies the `dispatchNotificationEvent()` method to its list of observers. It is the responsibility of each observer to override the `dispatchNotificationEvent()` method according to its own needs. The first step is to check whether the corresponding event should be handled or ignored and then perform the necessary updates.

The `IVBConnection` class overrides the `dispatchNotificationEvent()` method as follows:

```

IObserver&
IVBConnection::dispatchNotificationEvent(const INotificationEvent& anEvent)
{
    ...
    if (checkEvent(anEvent)) {
        try { perform (anEvent); }
        catch (IException exc&) {handleException(anEvent, exc);}
    }
    ...
}

```

Each connection derived from the `IVBConnection` class (and its descendants) must override:

- ❑ the `checkEvent()` method
- ❑ the `perform()` method

In our example, the `closingViewConn0` class redefines the `checkEvent()` and the `perform()` methods as follows:

```

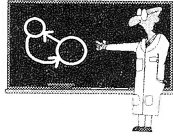
virtual int checkEvent (const INotificationEvent& anEvent) const
{
    return ( (anEvent.notificationId() == IPushButton::buttonClickId );
}

void perform (const INotificationEvent& anEvent)
{
    IFUNCTRACE_DEVELOP();
    ITRACE_DEVELOP ("firing... closePB(buttonClickEvent) to mainWindow(close)");
    target->close();
}

```

Since the `closingWindowConn0` instance is registered to the `iclosePB` observers list, the connection is said to be fired or activated, and the `perform()` method is called.

### Technical Information



Attribute-to-attribute connections are bidirectional. Thus, such a connection must monitor both the source and the target attributes. The IVBAttributeConnection overrides the dispatchNotificationEvent() method as follows:

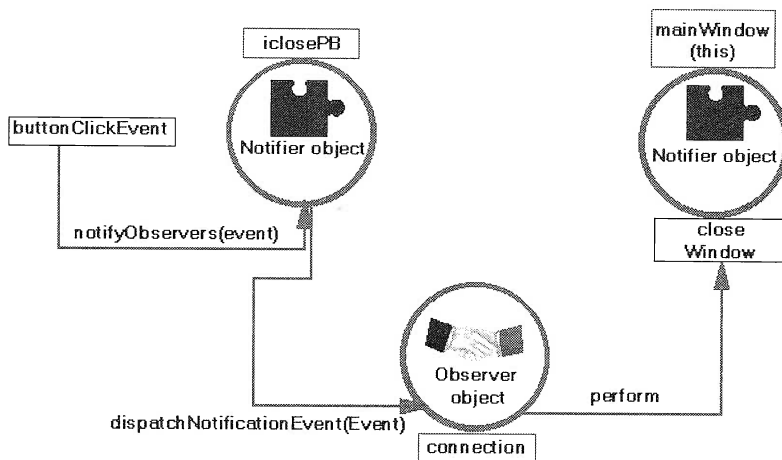
```

IObserver& IVBAttributeConnection::
dispatchNotificationEvent (const INotification-
Event& event)
{
    iStatus = (IVBConnection::Process) check-
Event(anEvent);
    if (iStatus == sourceChanged) {
        try { performSetTarget();}
        catch (IException& exc) { handleTargetEx-
ception(exc);}
        ...
    } // endif
    else {
        if (iStatus == targetChanged) {
            try {performSetSource();}
            catch (IException& exc)
{handleSourceException(exc) ;}
        } // endif
    } // endif-else
}

```

Any descendant from the IVBAttributeConnection class must therefore override the checkEvent(), performSetSource() and performSetTarget() methods.

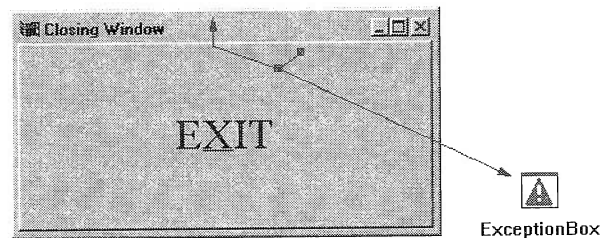
Figure 139 summarizes the notification flow.



**Figure 139.** Visual Builder Notification Flow

## Using Connections as Notifiers

Let us now modify our example: We want to bring up a message box whenever the connection cannot be fired and display the exception. We add an `IMessageBox*` part (see the `ExceptionsBox` part in Figure 140) to our example, and we draw an event-to-action connection (`Connection0`) between the `exceptionOccurred` event of the first connection (`Connection1`) and the `showException` action of the `ExceptionsBox` part.



**Figure 140.** Sample2 Window: Using a Message Box for Exception Handling

The code generated for the `connection0` definition is now different. The event-to-action `connection0` plays two roles: the role of observer to perform the closing of the `mainWindow` and the role of notifier to send the `exceptionOccurred` event to `connection1`. The class definition generated for `connection0` is now:

```
class closingWindowConn0 : public IVBConnectionPart {
};
```

The main difference between this sample and the previous sample in “Parts and Connection Initialization” on page 357 is that `connection1` has to register itself as an observer to a connection instead of a visual or nonvisual part, as shown in the following generated code:

```
class closingWindowConn1: public IVBConnection {
public:
    //-----
    // Constructors / destructors
    //-----
    closingWindowConn1(closingWindowConn0 * aSource, IMessageBox * aTarget) :
        source(aSource), target(aTarget)
    {
        handleNotificationsFor(*source);
    }
    ....
private:
    //-----
    // private member data
    //-----
};
```

```
        closingWindowConn0* source;  
        IMessageBox * target;  
};
```

Both connections are added to the connection list maintained by the closingWindow part.

```
closingWindowConn0 *conn0 = new closingWindowConn0(iclosePB, this);  
iclosingWindowConnectionList->add (conn0);  
closingWindowConn1 *conn1 = new closingWindowConn1(conn0, iExceptionBox);  
iclosingWindowConnectionList->add (conn1);
```

In addition, connection0 must enable itself for notification. This step is completed during the initialization of the connection, which is triggered by the following line of code:

```
iclosingWindowConnectionList->initialize();
```

The initialize() method is implemented as follows for the IVBConnectionPart class:

```
void IVBConnectionPart::initialize()  
{  
    if (iStatus == uninitialized) {  
        iStatus = normal;  
        enableNotification();  
    }  
}
```

As a notifier, connection1 must call the notifyObservers() function if an exception occurs while closing the window. This notification is sent by the handleException() function implemented in the IVBConnectionPart class. The handleException() function is called in the dispatchNotificationEvent() function (see the code on page 359):

```
int IVBConnectionPart::handleException(const INotificationEvent& event,  
                                       const IException& exc)  
{  
    notifyObservers(INotificationEvent(exceptionId,  
                                       *this,  
                                       true,  
                                       IEventData((void*) &exc)));  
    ....  
}
```

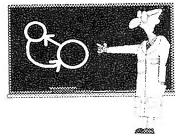
With all of the above in mind, you are ready to take your existing C++ classes and transform them into parts. The next section provides you with some guidelines to achieve that goal.

## From Classes to Nonvisual Parts in Visual Builder

In this section, we convert a generic flat file class to a fully enabled nonvisual part that can be used in Visual Builder. (See Appendix E on page 525 for the source code for the flat file class.) We provide step-by-step instructions for performing this task. We focus on nonvisual parts, but the method of creating your own visual parts is exactly the same, provided that you use the `IWindow` class as a parent for your visual part. Another solution would be to create a child class from any visual part, such as `IFrameWindow`, and modify it to suit your requirements.

To achieve the transition from a C++ class to a nonvisual part, you must enable your C++ class as a notifier and declare the part interface used in Visual Builder. First, modify the class definition so that your class inherits from `IStandardNotifier`. Then, create a unique notification ID for all events that your part might send. Finally, modify the code to call the `notifyObservers()` function each time an attribute is modified (if you want the change to be public, of course). You can describe the part interface by either writing a part information file or using the Part Interface Editor in Visual Builder.

### Technical Information



If you want to use a part as the *source* for a connection, it must be able to send notifications to other parts. Thus, you must complete all modifications as described in this section. However, if you only want to use a part as the *target* of a connection, only the creation of the part interface is mandatory, and the part does not have to be enabled as a notifier. Such a part is called a *class interface* part. An example of a class interface part is provided in “Creating a Class Interface Part from Your Event Handler Class” on page 243.

The `flatFile` class has the following interface:

- Member data
  - `fileName` (`IString`)
  - `currentLine` (`IString`)
  - `fileHandler` (`fstream`)
  - `fileIsOpened` (`Boolean`)
  - `eofReached` (`Boolean`)



□ Member functions

- `getFileName` (get accessor for the `fileName` attribute)
- `setFileName` (set accessor for the `fileName` attribute)
- `getCurrentLine` (get accessor for the `currentLine` attribute)
- `setCurrentLine` (set accessor for the `currentLine` attribute)
- `readLine`
- `readFile`
- `writeLine`
- `open`
- `close`

Before you actually transform a class into a part, ask yourself the following questions:

1. Which member functions should I include in the part interface?
2. Which attributes should I include in the interface, and do I have to send an event whenever the attributes undergo a state change?
3. Do I have to create additional events that would be useful when using this part in Visual Builder?

Typically, the `fileHandler`, `fileIsOpened`, and `eofReached` data members are internal to the behavior of the class and should not be used from Visual Builder. Rather, you want to signal when the `fileIsOpened` and `eofReached` Booleans are true. Therefore, you add the *fileOpened* and *eofReached* events to the part interface.

The only attributes that you have to modify or retrieve through their get and set member functions are `fileName` and `currentLine`.

Finally, you must declare the `open`, `close`, `readFile`, `readLine`, and `writeLine` member functions as actions.

With all of the above in mind, you can begin to write the part interface after taking some precautionary measures to protect your code.

## Protecting your Code

Before doing any manipulation to your code, we strongly recommend that you make a backup! The rationale for this is pretty straightforward: when you create your code you usually save it in files with extensions *.hpp* and *.cpp*. Those extensions are default extensions for Visual Builder, which implies that creating the flatfile part as explained above sets the default Visual Builder generation files to *flat-*

*file.cpp* and *flatfile.hpp*. Thus, choosing **Save and generate**→**Part source** will *erase* any code stored in those files. The following steps can help you protect your code:

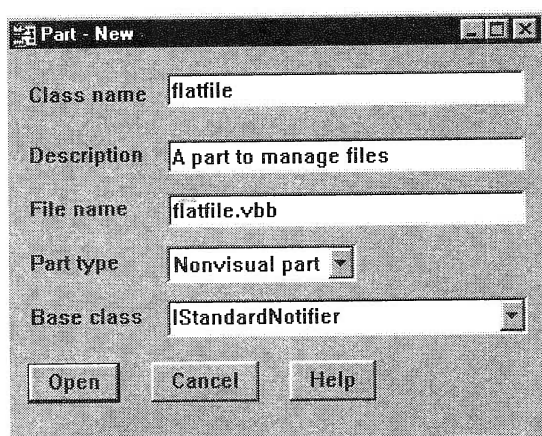
- 1). Rename your *.cpp* and *.hpp* files with another extension, for example *.cpu* and *.hpu*.
- 2). Create two dummy files, *flatfile.cpp* and *flatfile.hpp* as follows:

```
//-----
// flatfile.cpp
//-----
#include "flatfile.hpp"
#include "flatfile.cpv"
#include "flatfile.cpu"
//-----
// flatfile.hpp
//-----
#include "flatfile.hpv"
#include "flatfile.hpu"
```

Using this technique, your code is safe, even if you inadvertently choose the **Save and generate**→**Part source...** option. Files with extension *.hpu* and *.cpv* are used to store the notification IDs we need for each event or attribute.

## Creating the FlatFile Nonvisual Part

To create the nonvisual part, start Visual Builder and select **Part**→**New**. Figure 141 shows the information to enter for the *flatFile* part.



**Figure 141.** Part-New Window: Creating a FlatFile Nonvisual Part

Once this part is created, you must switch to the Class Editor and specify the appropriate values for the Code Generation files, as well as for the User Defined files. You must specify the Code Generation files to be `flatfile.cpp` and `flatfile.hpp` (to comply with our “Code Protection Policy”). You also must specify that the user-defined file names are `flatfile.hpv` and `flatfile.cpv`.

## Describing the Part Interface

You can describe a part interface in two ways. The technique you use depends on how much code you have and how complex it is:

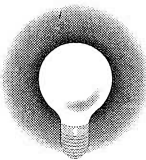
- ❑ Using the class browser, import the methods definitions of your C++ class into the Visual Builder Part Interface Editor and then create a VBB file.
- ❑ Write a part information file (a VBE file) and import it into Visual Builder to produce a VBB file.

### Using the Part Interface Editor

The first step is to generate browser database files for your existing code. The usual method is to compile the code with the `/Fb` option. You might also use the QuickBrowse facility of the browser, directly from Visual Builder. This facility is available from any Visual Builder editor provided that you started Visual Builder from a WorkFrame project.

To import the methods definition, switch to the Part Interface Editor and select **Browser→Open Browser Data**, Visual Builder looks for a `flatfile.pdb` file in the working directory (it actually builds this name from the nonvisual part name and the `.pdb` extension). If you specify **Browser→Quick Browse**, the Quick Browse action is started in the WorkFrame project. In both cases, Visual Builder populates the list of possible get and set accessors from the results of the browsing action. Using this technique can avoid the long, boring task of manually entering the definition of the functions by using the Part Interface Editor (and probably introducing a few errors).

#### Tip

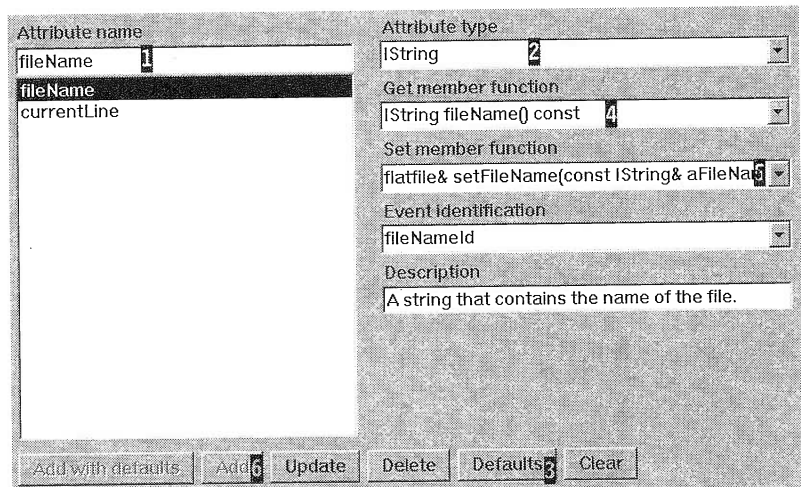


Browser data analysis is case-sensitive. In other words, if you call your part *flatFile* and your class name is *FlatFile*, Visual Builder will not be able to import the methods definition. To avoid any naming problems, a good rule is to give the same name to your class, the corresponding nonvisual part, and the files where you store your class definition.

You can now create your attributes, actions, and events in the same way as explained in “AMarketingInfo” on page 226. For example, the `fileName` attribute is defined as described in Figure 142. The easiest way to create the attribute definition is to enter the name for the attribute **1**, select its type **2**, click on the **Defaults** button **3** so that the notification ID is generated for you, and select the appropriate **get** **4** and **set** **5** methods (which are actually defined in your code). Then click on the **Add** **6** button to define the attribute.

Accessors are used by VisualBuilder to access attribute values (Visual Builder cannot access an attribute value directly). You must provide:

- ☐ A get accessor for each attribute
- ☐ A set accessor for each attribute that you may use as the *target* for a connection.



**Figure 142.** Part Interface Editor Window: Creating an Attribute Definition

Once you have described all of your part features, the next step is to generate the code that you have to add to your legacy code. You already have the methods definition as well as the attributes definition, so all you need is the events declaration and definition. Visual Builder appends this code to the user-defined generation files you defined in the Class Editor, namely, `flatfile.hpv` and `flatfile.cpv`.

To generate the code corresponding to events, select **Save and Generate→Feature source...** from the *File* menu item and select *only* the items from the events data members list. Then click on the **Generate selected** push button to generate the source code of the selected items. If the `flatfile.cpv` and `flatfile.hpv` files do not exist,

they are automatically created. If the files exist, Visual Builder appends the generated code to them. The events definition and declaration are as follows:

```
// Definition in the flatfile.cpv file

const INotificationId _Export FlatFile::fileOpenedId = "FlatFile::fileOpened";
const INotificationId _Export FlatFile::eofReachedId = "FlatFile::eofReached";
const INotificationId _Export FlatFile::fileNameId = "FlatFile::fileName";
const INotificationId _Export FlatFile::currentLineId = "FlatFile::currentLine";

// Definition in the flatfile.hpv file

static const INotificationId IVB_IMPORT fileOpenedId;
static const INotificationId IVB_IMPORT eofReachedId;
static const INotificationId IVB_IMPORT fileNameId;
static const INotificationId IVB_IMPORT currentLineId;
```

***Do not forget to save your part but do NOT generate the code for it!!!***

### ***Writing a Part Information File***

A part information file, often referred as to a Visual Builder Export (VBE) file is a flat file that describes a part interface. It contains the definition of the part itself, as well as its actions, attributes, and events. Visual Builder can import a VBE file to create the corresponding VBB file. For details on the syntax of the statements you use to write such a file, see *Building VisualAge C++ Parts for Fun and Profit*.

Here is an extract of the export file that you must create to import the FlatFile part in Visual Builder:

```
//VBBeginPartInfo: FlatFile,"A Flat File Management Class"
//VBParent: IStandardNotifier
//VBIncludes: "FlatFile.hpp" _FLATFILE_
//VBPartDataFile: FlatFile.vbb
//VBComposerInfo: nonvisual
//VBEvent: fileName, "fileName",fileNameId
//VBAttribute: fileName,
//VB:      "A string that contains the file name",
//VB:      IString,
//VB:      IString fileName() const,
//VB:      FlatFile& setFileName(const IString& aFileName),
//VB:      fileNameId
//VBAction: close
//VB:      "Close the flat file",,
//VB:      FlatFile& close()
//VBPreferredFeatures: open, close, fileName, currentLine
//VBEndPartInfo: FlatFile
```

This method is quite efficient in terms of time and simplicity if you have a class with few methods and few attributes. You can also use a language such as REXX to write templates for part information files and develop a script that would generate the export file for you from the class information.

## Modifying Your Code

To use your part as a *source* for a connection, you must modify it to become a notifier. Modify its definition as follows:

```
class FlatFile: public IStandardNotifier {
public:...
```

Because your part uses the IBM Open Class Library notification framework, you must include the corresponding definition files as well as the files generated from the Part Interface Editor. Include the following files in the flatfile.cpp file:

- ☐ inotifev.hpp (NotificationEvent class Definition)
- ☐ istdntfy.hpp (IStandardNotifier class Definition)

Now your part must send a list of events to the “external world.” Therefore, you must call the notifyObservers() method each time an attribute of your part is modified or whenever you want to signal a particular event. For example, whenever the fileName attribute of your FlatFile part changes, you call the notifyObservers() method by modifying the setFileName() method as follows:

```
FlatFile& FlatFile::setFileName(const IString& aFileName)
{
    if ( fileName != aFileName ) {
        fileName = aFileName;
        notifyObservers(INotificationEvent(FlatFile::fileNameId, *this));
    }
    return *this;
}
```

### Tip



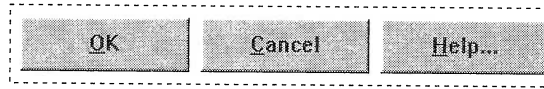
To avoid unnecessary overhead, we highly recommend testing the new value to be assigned to an attribute *before* assigning the value and calling the notifyObservers() function.



Congratulations! Your class is now a part and you can fully exploit its power from Visual Builder.

## When Parts Become Observers...

Let us take a simple example of a part becoming an observer. With the Composition Editor, create a `defaultButtons` visual part from an `ICanvas`\* part and three `IPushButton`\* subparts as depicted in Figure 143.



**Figure 143.** The `defaultButtons` Composite Visual Part

The `defaultButtons` visual part can be reused in other visual parts to provide a standard look and feel in your applications. For this purpose, you must be able to access the features of the `defaultButtons` subparts from any of the parts where you reuse the `defaultButtons` part. You know by now that this is not possible unless you promote the features of the subparts, that is, the features of the three push buttons. Let us promote the `buttonClickEvent` feature of each push button and examine the generated code.

The first thing you notice when you look at the code is that the `defaultButtons` part inherits from both the `ICanvas` and `IObserver` classes:

```
class defaultButtons : public ICanvas, public IObserver {
public:
    ...
};
```

In fact, the `defaultButtons` part must be able to observe the subparts whose features are promoted and catch the notification events corresponding to the promoted features. To publish those events, Visual Builder generates the `defaultButtons` source code so that one notification event ID is created for each promoted feature, and each promoted feature is added to the `defaultButtons` public interface:

// Declaration in the `defaultb.hpp` file

```
static const INotificationId IVB_IMPORT pBOkButtonClickEvent;
static const INotificationId IVB_IMPORT pBCancelButtonClickEvent;
static const INotificationId IVB_IMPORT pBHelpButtonClickEvent;
```

// Definition in the `defaultb.cpp` file

```
const INotificationId _Export defaultButtons::pBOkButtonClickEvent =
    "defaultButtons::pBOkButtonClickEvent";

const INotificationId _Export defaultButtons::pBCancelButtonClickEvent =
    "defaultButtons::pBCancelButtonClickEvent";

const INotificationId _Export defaultButtons::pBHelpButtonClickEvent =
    "defaultButtons::pBHelpButtonClickEvent";
```

To observe its subparts, the `defaultButtons` part must register itself to the list of observers of each subpart that has a promoted feature. For this purpose, Visual Builder generates a call to the `handleNotificationFor()` method in the `initializePart()` method of the `defaultButtons` part for each subpart that has a promoted feature:

```
defaultButtons & defaultButtons::initializePart()
{
    this->handleNotificationsFor(*iPBOK);
    this->handleNotificationsFor(*iPBCancel);
    this->handleNotificationsFor(*iPBHelp);
    makeConnections();
    notifyObservers(INotificationEvent(readyId, *this));
    return *this;
}
```

Visual Builder also generates one member function for each subpart that has at least one promoted feature. This member's feature returns the subpart itself and is used when the `defaultButton` is reused in another part to access the subpart features:

```
IPushButton * getPBOK() const { return iPBOK };
IPushButton * getPBCancel() const { return iPBCancel };
IPushButton * getPBHelp() const { return iPBHelp };

```

As an observer, the `defaultButtons` part must override the `dispatchNotificationEvent()` virtual method. Basically, the `defaultButtonsPart` maps the notification event published by its subparts to the new notification events created for each promoted feature. As an example, the following piece of code captures the `buttonClickId` from the `OK` push button and publishes it under its new name:

```
IObserver&
defaultButtons::dispatchNotificationEvent(const INotificationEvent & anEvent)
{
    if ((anEvent.notificationId() == IPushButton::buttonClickId)
        &&
        (iPBOK == &anEvent.notifier()) )
        notifyObservers(INotificationEvent(pBOKButtonClickEvent,
                                           *this,
                                           anEvent.hasNotifierAttrChanged(),
                                           IEventData((void *)anEvent.eventData()),
                                           anEvent.observerData()));
    else
        ...
}
```



Congratulations! You are now experts on the IBM notification framework and its application in Visual Builder.





# 11

## More about Data Access Builder...

*Never change a running program.*

-Proverb of programmers

In Chapter 6, “Mapping Relational Tables Using Data Access Builder,” on page 131 you map the REAL database to parts directly using the embedded SQL method, integrate the Data Access Builder with the other visual and nonvisual parts, and build a whole application. Windows NT is the development platform; the Single-User version of DB2 for Windows NT provides your application with a database access, and REAL is created as a local DB2 database. Notice that the application has also been fully tested on Windows 95 with the Single-User version of DB2 for Windows 95.

In this chapter, you learn more about using the Data Access Builder. First, you access the DB2 database in a client/server approach with DB2 Client Access Enabler (CAE) for Windows 95. Then you access the database with the ODBC access method instead of embedded SQL. Finally, you recreate the REAL database as dBase files (dbf) and access them again with the ODBC access method. Thus, we show you

step by step how you can migrate from a full-blown development environment with Windows NT and DB2 Single-User for Windows NT to a smaller environment with Windows 95 and flat files.

In this chapter, you also learn how to enhance the data access in the application, link the container contents to a subset of all the rows in the table, define ordering information for the data, customize the way your data are displayed, and benefit from advanced DB2 features to reduce your code.

## Accessing the Database from a Windows 95 Client

Our initial environment setup, Windows NT and DB2 for Windows NT Single-User, represents a typical development platform. In the further sections we consider an application environment that reflects a client/server approach; that is, the application runs on the workstation platform (the client), and the data is stored on the database server. Of course, you can also use this environment for development. You might even need it for debugging and testing your application.

DB2 database servers are available on a broad range of systems (see Figure 144): MVS/ESA, OS/400, VM, VSE, UNIX-based platforms (AIX, HP-UX, Solaris, and Sinix), OS/2, Windows NT, and Windows 95. Remote clients can also run in a variety of operating environments, including OS/2, several UNIX-based environments, DOS, and Windows. Along with the DB2 Server comes a Client Pack, which includes platform-specific versions of the DB2 CAE component for all those environments.

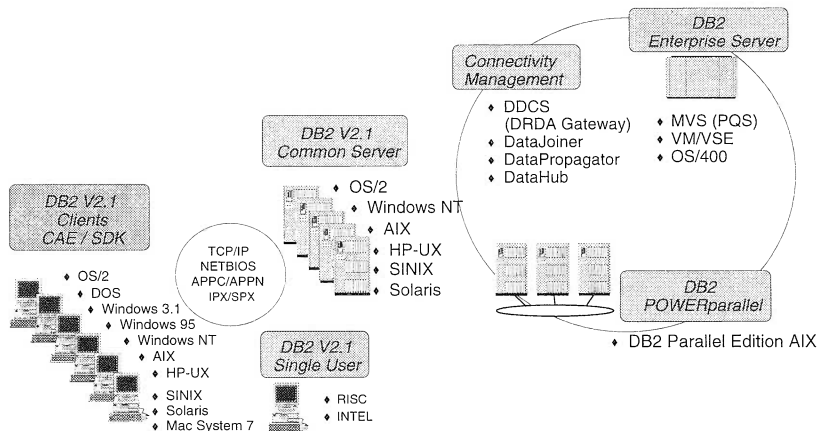


Figure 144. DB2 Platforms

A workstation that has DB2 CAE installed can access any DB2 server, using any of a number of communication protocols. DB2 Server for Windows NT supports TCP/IP, NetBIOS, and IPX/SPX with all Windows versions of DB2 CAE. In addition, the APPC protocol is supported for OS/2 and AIX clients. Refer to the *DB2 Information and Concepts Guide* for more information about DB2.

To access the database from a Windows 95 client, you first set up the database server on Windows NT and next you install the Windows 95 client. Below we discuss both procedures as well as the impact on the application development.

## Setting Up the Database Server

You set up the database server in three steps. First, you install DB2 and configure it to start automatically at boot time. Then, you create the REAL database manually. Finally, you configure the database server for remote connection, using NetBIOS or TCP/IP protocol.

If you want to establish a remote connection with another protocol, refer to the *DB2 for Windows NT Administration Guide*.

### ***Installing DB2 for Windows NT***

For information about installing the product, refer to the *DB2 for Windows NT Installation and Operation Guide*. The installation adds two services to Windows NT: the DB2 Security Server and the DB2 engine.

The DB2 Security Server handles security checking. To access a database, you must pass three security checks:

1. Authentication
2. Database connection
3. Database object usage

*Authentication* is performed outside DB2 by the Windows NT security services that validate your user ID and password. A DB2 instance can be configured with server or client authentication for remote connections to its databases (for more information concerning authentication you can also refer to “DB2 for Windows Authentication” on page 336). Server authentication is the default. In this case, and also if the Windows NT server considers the Windows 95 client as an untrusted client, user ID and password must be specified when connecting to the database and are validated on the remote Windows NT server.

In the process of *database connection*, DB2 verifies whether you have sufficient privileges to gain access to the database. Additional security checking is performed each time you want to use a data object.

After installation you can simply start the DB2 engine with the **db2start** command. However, you probably want DB2 and DB2 Security Server to start automatically with the Windows NT server:

1. Open the Control Panel window.
2. Open Services window.
3. Select **DB2 - DB2** and click on **Startup**.
4. Select **Automatic** as Startup Type and click on **OK**.
5. Select **DB2 Security Server** and click on **Startup**.
6. Select **Automatic** as Startup Type and press **OK**.
7. Close the Services and Control Panel windows.

You also have to add a new user, **userid**, with the password, **password**, to the Windows NT server or domain. This user does not need any special user rights for Windows NT. You grant the DB2 privileges for accessing the REAL database at the next step, while you create the database.

### ***Creating the REAL Database***

You do not have to manually create the database on your Windows NT development workstation; instead you take advantage of the installation routine. In this section, we reveal the secrets of the installation routine and show you how to create the REAL database, fill it with some default data, and grant access to the user ID used in the application.

On the CD-ROM that accompanies this book, there are four files with an extension of .ddl, that contain DB2 commands to set up the REAL database. You use the creatab.ddl and update.ddl files to create all tables in the database and fill them with some default data.

To set up the database, follow these steps:

1. Open a DB2 command window.
2. Switch to the drive containing the CD-ROM.
3. Start DB2:

**db2start**

4. Create a new database REAL on a local drive with sufficient space:

**db2 create real on c**

Be patient, this will take some time.

5. Connect to the database:

**db2 connect to real**

6. Create the tables and views:

**db2 -tf creatab.ddl**

7. Fill the tables with default data:

**db2 -tf update.ddl**

8. Grant full administration privileges on the REAL database to user userid:

**db2 grant dbadm on database to user userid**

9. Terminate the database connection

**db2 terminate**

### ***Configuring the Remote Connection for NetBIOS Protocol***

The NetBIOS protocol identifies computers by a unique NetBIOS name, also called the *computer name*. You find the name of your computer in the Network object of the Windows Control Panel. Windows 95 displays the computer name in the Identification tab page.

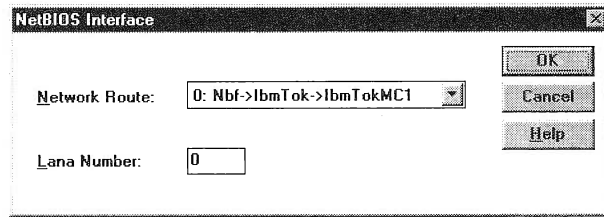
Suppose that you use colorado as the computer name of the database server. Then you specify this NetBIOS name to DB2 by issuing the following command in a DB2 command window:

**db2 update dbm cfg using nname colorado**

Windows NT passes NetBIOS packets through NetBEUI using the NetBIOS Frame (NBF) protocol driver. If you install both NetBIOS and TCP/IP protocols, Windows NT you allows also to pass NetBIOS over TCP/IP (NBT). You should verify that you select NBF to use the logical LAN adapter card 0:

1. Open **Control Panel**.
2. Open **Network** window.
3. Select **NetBIOS Interface** and click on **Configure**.
4. Verify that the route that starts with Nbf uses the Lana number 0 (see Figure 145) and the one starting with NetBT uses the Lana number 1.

5. Click on **OK** for the Network Interface and Network windows



**Figure 145.** Configuring the NetBIOS Interface

Finally, set the DB2COMM variable in the system environment, so that DB2 can enable remote connections through NetBIOS:

1. Open the **Control Panel** window.
2. Open the **System** window.
3. Select any variable in the list of system environment variables to ensure that the new variable, DB2COMM, will be added in the system environment.
4. Type **DB2COMM** as variable name.
5. Type **NETBIOS** as value.
6. Click on **Set** to define the new variable.

Click on **OK** to close the window and save the changes.

### ***Configuring the Remote Connection for TCP/IP***

We assume that a TCP/IP domain is set up in your environment. You can verify it and find your host name:

1. Open the **Control Panel** window.
2. Open the **Network** window.
3. Select **TCP/IP Protocol** and click on **Configure**.
4. Click on **DNS**.

If there is an entry in the list of DNS Servers, a TCP/IP domain is configured. You can also see the host name of your computer.

If you do not have a TCP/IP domain, you can easily install TCP/IP on Windows NT and Windows 95. For each platform, specify a unique IP address in the TCP/IP configuration. You also add a HOSTNAME variable to the system environment and provide a host name, for example DB2SRV.

The database server requires a TCP/IP port (number *n*) for communication with remote clients. You should also reserve a second port immediately after the first port (number *n*+1) for the interrupt requests of down-level clients. For example, the current DB2 Client Access Enabler versions for DOS and Windows 3.1 are not at the same version-level as DB2 for Windows NT. If you omit the interrupt port, DB2 will append an error message in the `db2diag.log`, each time you start it. You define the TCP/IP ports in the `services` file located in the `system32\drivers\etc` subdirectory of the Windows NT system root directory. The file is ordered by port numbers; insert the lines with unused port numbers at the appropriate location such as:

<code>db2cp</code>	<code>3700</code>	<code>#DB2 port for communication</code>
<code>db2ip</code>	<code>3701</code>	<code>#DB2 port for interrupt req from down-level clients</code>

To inform DB2 about these communication ports, update the service name, `svcename`, in the DB2 configuration. Start a DB2 command window and issue the following command:

### **db2 update dbm cfg using svcename db2cp**

Finally, set the `DB2COMM` variable in the system environment, so that DB2 can enable remote connections through TCP/IP:

1. Open the **Control Panel** window.
2. Open the **System window**.
3. Select any variable in the list of system environment variables to ensure that the new variable, `DB2COMM`, will be added in the system environment.
4. Type **DB2COMM** as variable name.
5. Type **TCPIP** as value.  
If you already set `DB2COMM` for NetBIOS, change the value to `NET-BIOS,TCPIP` to enable both.
6. Click on **Set** to define the new variable.
7. Click on **OK** to close the window and save the changes.
8. Shut down and restart the database server.

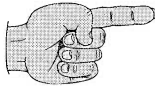
Your database server is now ready to receive requests from remote clients either with TCP/IP or NetBIOS.



## Setting Up the Windows 95 Client

Install the DB2 CAE for Windows 95 according to the installation instructions that come with DB2 for Windows NT. If you use a dual boot configuration and have already installed DB2 for Windows NT, use a different installation path for DB2 CAE.

### Read This



If you intend to build the application under Windows 95, you need to install the DB2 Software Developer's Kit instead of the DB2 CAE. The kit includes precompiler, header files, and link libraries.

To access the database server with NetBIOS, define the computer name of your Windows 95 client to DB2 CAE and catalog a node name for the database server. It is a good convention to use the computer name as node name. Supposing that the computer name of your Windows 95 client is *napa*, perform the following steps:

1. Open a DB2 command window.
2. Specify the computer name to DB2:

**db2 update dbm cfg using nname napa**

3. Add a node for the database server:

**db2 catalog netbios node colorado remote colorado adapter 0**

We assume that you use the LAN adapter card 0 for the NetBIOS protocol.

To access the REAL database remotely with TCP/IP, you need to:

1. Ascertain TCP/IP host name resolution for the database server.
2. Reserve a TCP/IP port for DB2 communication.
3. Add a new node to the node directory.
4. Add a new remote database to the database directory.

We detail these steps in the following paragraphs.

When a connection to a remote database is requested, DB2 CAE tries to resolve the host name that is cataloged as the node for the remote database. The Domain Name Server, often referred to as *DNS*, provides the service for host name resolution. If you do not have a TCP/IP domain, create a local hosts file in the Windows 95 root directory, *C:\Windows* by default. You can use the existing *hosts.sam* file as a template. Add a line for the host name of your database server. The IP address and the host name must match the settings in the TCP/IP configuration and *HOSTNAME* environment variable of the database server:

127.0.0.5      DB2SRV    # The DB2 database server

As you did with the database server, insert a line for the DB2 communication port to the existing services file located in the Windows 95 root directory. DB2 CAE for Windows 95 does not require the interrupt port.

Add a new node entry to the DB2 node directory and specify the node name, TCP/IP host name, and communication port for this node. If you already specified the node name *colorado* for NetBIOS, you must use another one for TCP/IP. In addition, add a new remote database in the DB2 database directory and define the node where the database is located:

1. Open a **DB2 command window**.

2. Add the node:

**db2 catalog tcpip node *hostname* remote *hostname* server *db2cp***

3. Add the remote database:

**db2 catalog database real at node *hostname***

4. You can now connect to the REAL database:

**db2 connect to real user *userid* using password**

You can set up a Windows NT workstation the same way. Before you can catalog the remote database, you have to remove any previously defined database with the same name. You do so by issuing the command `db2 drop database db-name` for local databases and `db2 uncatalog database db-name` for remote databases. Local and remote databases can be equally used, once they are cataloged in the DB2 database directory. Thus, you can easily switch from local to remote databases.

Your Windows 95 client is now ready to establish remote connection to the REAL database on the database server.

## Impact on the Application Development

You might wonder whether you can simply restart the application. The answer is “yes, sort of.” You do not have to rebuild the application, but you do have to bind it to the newly created database. Applications using embedded SQL must be explicitly bound to the database they access.

The `bind.ddl` file on the CD-ROM contains the bind commands for all generated data access classes. You also have to bind the Data Access Class Library explicitly: the required bind file, `cppwac12.bnd`, is in the `\ibmcppw\bnd` subdirectory of VisualAge for C++ for Windows.

The reason you need not perform these binds for the local database is that when you connected to the database from within the Data Access Builder, it did the job for you. And at the time you built the Dacslib project, the precompilation of the generated classes bound them to the local database.

Intentionally, we had you grant administration privileges on the whole REAL database to user `userid`. Even though table privileges would be sufficient to run the application, you need additional privileges as user `userid` to perform a bind command. And you use this user `userid` to connect to the database from the Windows 95 client:

1. Open a **DB2 command** window.
2. Switch to the working directory of the dacslib project:  
**cd \real\book\dacslib\win**
3. Connect to the remote database:  
**db2 connect to real user userid using password**
4. Bind your application specifying the appropriate drive for your CD-ROM:  
**db2 -tf e:\bind.ddl**
5. Bind the Data Access Class Library:

**db2 bind d:\ibmcppw\bnd\cppwac12.bnd blocking all grant public**

Row blocking reduces database manager overhead by retrieving a block of rows in a single operation, reduces the network traffic for remote database and speeds up cursored operations.

Congratulations! Your application can now run under Windows 95 and access the REAL database on the database server.



## A Short History of SQL Data Access

In this section we give you a short summary of different data access methods. With this information you are prepared to change the data access method used in the application.

SQL provides language elements to manage databases, access, and manipulate the database data. It is not a full programming language in itself. You use it together with a common programming language like C++.

The concept of relational databases bases on a mathematical theory, developed in the IBM research centers in the early 1970s. The theoretical work *A Relational Model of Data for Large Shared Data Banks* of E.F. Codd outlines the basics of relational models in 1970. D.D. Chamberlain et al., *SEQUEL*, 1974 describes the concept of a *Structured English Query Language*, the predecessor to SQL. In the mid-1970s IBM started a development project and created the first relational database SYSTEM/R. After tests in 1979, IBM initiated the development of a commercial system and announced SQL/DS for the operating system DOS/VSE in 1981. Two years later, SQL/DS for VM/CMS and DB2 for MVS followed.

In 1982, an American National Standards Institute (ANSI) committee was initiated to standardize the SQL language. The first version (ANSI X3.135) was passed in 1986.

DB2 for Windows NT supports three data access methods: Embedded SQL, Microsoft ODBC specification, and DB2 Call Level Interface to integrate the SQL language with your programming language.

## Embedded SQL

Embedded SQL is the most popular programming interface for accessing databases. It is defined in the SQL ANSI 1989 standard as one of three ways to integrate the SQL language into your code.

It enables you to write SQL statements in a host language such as C++ or COBOL. An application that includes embedded SQL statements requires a precompiler to convert the statements into equivalent host language source code. After precompiling, you compile, link the modified source code, and bind it to a particular database.

The ANSI 1989 standard defined only static SQL statements, whose syntax is fully known at precompile time, when the statement is prepared and an executable form is stored in the package file. The ANSI 1992 standard introduced dynamic SQL statements, which are prepared at run-time. After preparation, the executable form is cached and accessible, as long as it is not replaced and the database connection is established.

Dynamic and static SQL have both advantages and disadvantages. Developers are most interested in the performance aspect. The efficiency of dynamic and static SQL depends on the situation. On one hand, dynamic SQL has the overhead of the preparation step before

execution. This overhead must be set in relation to the total time to run the SQL statement. For a long, time-consuming statement the preparation time adds comparatively less time, as for a small statement that executes fast. If a statement is executed many times, the overhead is multiplied by the number of repetitions. But caching the executable form causes the preparation to be performed only once and thus reduces the total overhead.

On the other hand, the dynamic preparation has the advantage that it uses accurate data statistics to best optimize the execution of the SQL statement. When you have a choice, here are some of the considerations you should include, and a recommendation on whether to choose static or dynamic SQL:

<b>Table 67.</b> Comparing Static and Dynamic SQL	
<b>Consideration</b>	<b>Likely best choice</b>
Time to run the SQL statement <input type="checkbox"/> Less than 2 seconds <input type="checkbox"/> 2 to 10 seconds <input type="checkbox"/> More than 10 seconds	<input type="checkbox"/> Static <input type="checkbox"/> Either <input type="checkbox"/> Dynamic
Data uniformity <input type="checkbox"/> Uniform data distribution <input type="checkbox"/> Slight nonuniformity <input type="checkbox"/> Highly nonuniform distribution	<input type="checkbox"/> Static <input type="checkbox"/> Either <input type="checkbox"/> Dynamic
Range (<, >, BETWEEN, LIKE) predicates <input type="checkbox"/> Very infrequent <input type="checkbox"/> Occasional <input type="checkbox"/> Frequent	<input type="checkbox"/> Static <input type="checkbox"/> Either <input type="checkbox"/> Dynamic
Repetitious execution <input type="checkbox"/> Runs many times (10 or more times) <input type="checkbox"/> Runs a few times (less than 10 times) <input type="checkbox"/> Runs once	<input type="checkbox"/> Either <input type="checkbox"/> Either <input type="checkbox"/> Static
Nature of query <input type="checkbox"/> Ad hoc <input type="checkbox"/> Permanent	<input type="checkbox"/> Dynamic <input type="checkbox"/> Either
Frequency of statistics update <input type="checkbox"/> Very infrequently <input type="checkbox"/> Regularly <input type="checkbox"/> Frequently	<input type="checkbox"/> Static <input type="checkbox"/> Either <input type="checkbox"/> Dynamic
<b>Note:</b> "Either" means that there is no advantage to either static or dynamic SQL	

Static SQL provides better security. An application has the privileges of the authorized person who performed the explicit binding. These may differ from the actual user of the application. Thus, you allow a user to perform privileged actions only through the application. With dynamic SQL, the user must have the appropriate privileges granted; otherwise, he could bypass the application and misuse these privileges.

By nature, dynamic SQL provides you with the flexibility to construct SQL statements at run time. You often use this flexibility to offer your end-users a rich and flexible search capability.

## Call Level Interface

Like embedded SQL, a call-level interface (CLI) defines a mechanism for invoking SQL statements from within a host language. It is an application programming interface (API) that uses function calls to pass SQL statements. It supports dynamic SQL only, but it does not require a precompiler and explicit binding of the application. Thus, it increases the portability of database applications across different database servers.

The X/Open Company and the SQL Access Group jointly developed a specification for a callable SQL interface referred to as the X/Open Call Level Interface. Most of the X/Open CLI specification has been accepted as part of the ISO Call Level Interface Draft International Standard (ISO CLI DIS).

The Microsoft ODBC specification and IBM DB2 CLI are two data access methods that are based on the X/Open CLI and exploit the concept of a call level interface.

### ***Microsoft ODBC Specification***

Microsoft developed a callable SQL interface called Open Database Connectivity (ODBC) for Microsoft Operating Systems. ODBC 2.0 is based on the X/Open CLI 92 standard.

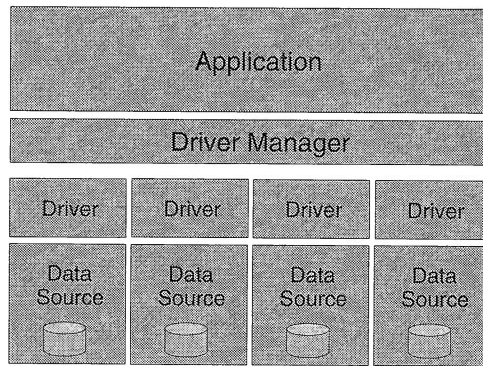
The ODBC specification also includes an operating environment where database-specific ODBC drivers are dynamically loaded at run-time. Therefore, it defines four components (see Figure 146):

**ODBC Application** performs the ODBC function calls to submit SQL requests and retrieve results. The application is linked to the Driver Manager dynamic link library (DLL).

**Driver Manager** is a DLL that provides the ODBC interface to the application, that is, it provides all entry points for ODBC functions. It maps data source names to the corresponding ODBC driver and performs the dynamic load and unload of the ODBC drivers.

**ODBC Driver** is a DLL that implements the ODBC function calls and interacts with the particular database. It establishes a connection to the data source, submits the SQL requests to the data source, performs the data conversion, and maps the return messages.

**Data Source** consists of data, its associated database management system, the DBMS platform, and the network to access the platform.



**Figure 146.** The ODBC Components

The concept of the ODBC Driver Manager provides your application with more flexibility. You link it once to the Driver Manager and the application can still access different DBMS. The price you pay is the Driver Manager's overhead, because all function calls to the ODBC Driver are passed through the Driver Manager.

Note that the term *data source* has a rather broad scope. A data source can be a single text file or a complex database with hundreds of tables managed with DB2 for MVS/ESA running on a System/390 Parallel Sysplex using the Distributed Relational Database Architecture (DRDA) protocol.

The ODBC interface faces the broad range of possible data sources and defines API and SQL conformance levels for ODBC drivers. A Level 1 API-compliant driver also provides functions, to retrieve such capabilities as supported data types and ODBC functions.

You can define a certain conformance level as a requirement for your application, reduce application features by using only a common denominator of ODBC functions or dynamically check the function of the actual ODBC driver.

With VisualAge for C++ for Windows NT Version 3.5, the Data Access Builder requires 32-bit ODBC 2.0 drivers that are full Level 1 API-compliant and support the following Level 2 API functions:

- ☐ SQLDataSources
- ☐ SQLForeignKeys
- ☐ SQLPrimaryKeys
- ☐ SQLTablePrivileges

However, the official support of IBM is extended to Sybase System 10, Oracle 7, and DB2 only, using ODBC CLI access and the ODBC drivers of these DBMS. Nevertheless, in this chapter you enhance your application and provide access to flat files in addition to the DB2 access.

### ***DB2 Call Level Interface***

The DB2 Call Level Interface is IBM's CLI to the DB2 family of database servers. The API is available for C and C++. It is based on the Microsoft ODBC and the X/Open CLI specifications and provides additional extensions to exploit specific DB2 features. DB2 CLI is also aligned with the emerging ISO CLI standard.

DB2 CLI includes support for all ODBC Level 1 functions, and all but three of the ODBC Level 2 functions. The unsupported functions are:

- ☐ SQLBrowseConnect
- ☐ SQLDescribeParam
- ☐ SQLSetPos

The DB2 features that are available to DB2 CLI applications, but not to ODBC applications, are:

- ☐ Large objects stored in the database
- ☐ SQLCA for detailed DB2-specific diagnostic information
- ☐ Null termination of output strings

Applications written explicitly for the DB2 CLI link directly to the DB2 CLI load library. ODBC applications access the ODBC Driver Manager that dynamically loads the DB2 CLI load library as an ODBC driver.



The DB2 CLI interface has several key advantages over embedded SQL:

- ❑ It is ideally suited for a client/server environment, in which the target database is not known when the application is built. It provides a consistent interface for executing SQL statements, regardless of which database server the application is connected to.
- ❑ It increases the portability of applications by removing the dependence on the precompiler. Applications are distributed as compiled applications or runtime libraries, and not as embedded SQL source code, which must be preprocessed for each DB2 database product, including DRDA.
- ❑ Individual DB2 CLI applications do not need to be bound to each database; only bind files shipped with DB2 CLI need to be bound (once) for all DB2 CLI applications. This can significantly reduce the amount of management required for the application as soon as it is in general use.
- ❑ DB2 CLI applications can connect to multiple databases, as well as multiple connections to the same database, all from the same application. Each connection has its own commit scope.
- ❑ DB2 CLI eliminates the need for application controlled, often complex global data areas, such as the SQLDA and SQLCA, typically associated with embedded SQL applications. Instead, DB2 CLI allocates and controls the necessary data structures, and provides a handle for the application to reference them. By eliminating global data areas, and associating all such data structures that are accessible to the application with a specific handle, DB2 CLI enables the development of multithreaded thread-safe applications where each thread can have its own connection and a separate commit scope.
- ❑ DB2 CLI offers enhanced parameter input and fetching capability, allowing arrays of data to be specified on input, retrieving multiple rows of a result set directly into an array, and executing statements that generate multiple result sets.
- ❑ DB2 CLI supplies a consistent interface to query catalog information (Tables, Columns, Foreign Keys, Primary Keys, etc.) contained in the various DBMS catalog tables. The result sets returned are consistent across DBMS. This shields the application from catalog changes across releases of database servers, as well as catalog differences among various database servers; thereby preventing applications from writing version-specific and server-specific catalog queries.
- ❑ Extended data conversion is also provided by DB2 CLI, requiring less application code when converting information between various SQL and C data types.
- ❑ DB2 CLI provides the ability to retrieve multiple rows and result sets generated from a stored procedures call.

Your decision in favor of one of these data access methods depends on the specific situation for your application. The decision is determined by the importance of such criteria as

- ☐ Flexibility to select DBMS at installation or run-time
- ☐ Portability across DBMS systems
- ☐ Performance of the application
- ☐ Encapsulation and security
- ☐ Conformance to standards
- ☐ Existing skills
- ☐ Usage of specific DBMS features

The Data Access Builder hides most of the differences among the three data access methods. It allows you to easily switch from one to another. We do not cover the DB2 CLI access separately, as it is rather similar to the ODBC data access method from a development point of view. Instead, we use the ODBC data access method, which obliges us to register the database to the ODBC Driver Manager.

## Using ODBC as Data Access Method

In this section, you switch the data access method used in the REAL application from embedded SQL to ODBC. You add the REAL database to the ODBC Driver Manager, generate the data access classes for this ODBC data source and rebuild the whole application.

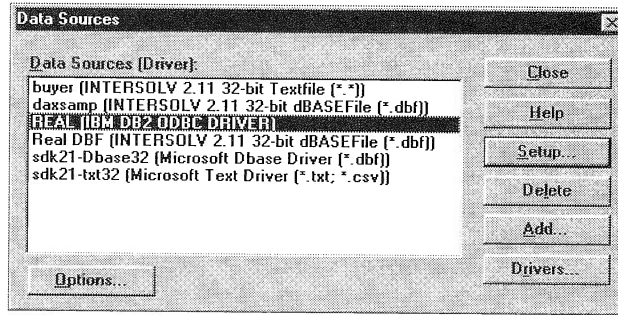
DB2 for Windows NT and the DB2 CAE versions for Windows 3.1x, Windows NT and Windows 95 include an ODBC driver that allows ODBC applications to access IBM DB2 family databases. The installation program copies the driver to the installation directory, but the driver is not added to the ODBC Driver Manager automatically. To add the driver, start the **ODBC Installer** located in the DB2 program folder. You should perform the installation after starting Windows, before any application that uses the ODBC support is started.

The ODBC Administrator is located in the Windows Control Panel and allows you to easily manage the data sources and ODBC drivers defined in your system. To add the REAL database as an ODBC data source, perform the following steps:

1. Open the **Control Panel** window.
2. Open the **ODBC Administrator** window.
3. Click on the **Add** to open the Add Data Source dialog box.
4. Select the **IBM DB2 ODBC Driver** and click on **OK**.
5. Select the **REAL** database in the combo box, provide a meaningful description of the data source, and click on **OK**.

6. Close the **ODBC Administrator** and the **Control Panel** windows.

The list of data sources contains the new REAL data source (see Figure 147).



**Figure 147.** Data Sources Window

## Differences in the Data Access Classes

The Data Access Builder creates the same set of classes for a table T regardless of the data access method used:

- ☐ T
- ☐ TDataId
- ☐ TDatastore
- ☐ TManager
- ☐ TManagerTemplate
- ☐ TManagerBase

Of course, the generated classes differ in their particular implementations. There are also minor changes in their interface, which you have to be aware of. The most important change lies in TDatastore, which is derived from, respectively,

- ☐ IDatastore, for embedded SQL
- ☐ IDatastoreDB2, for DB2 CLI
- ☐ IDatastoreODBC, for ODBC

These three classes are derived from the same base class, IDatastoreBase. IDatastore adds only one attribute feature to those of the IDatastoreBase part:

- ☐ shareModeExclusive, specifies exclusive access to the database.

IDatastoreDB2 and IDatastoreODBC add the same attribute features to the base part:

- ❑ `accessMode` specifies the access mode that is read-only or updateable.
- ❑ `autoCommit` specifies the autocommit mode.
- ❑ `connectString` specifies the connect string that is passed to the ODBC drive for a connect request.
- ❑ `driverPrompt` specifies when the user is prompted for more information to connect to the database.
- ❑ `isolationLevel` specifies the isolation level for the database.
- ❑ `connectUsingString` connects to a data source using the specified data source name, user ID, password, and connect string.

We do not exploit these features in the application, so the differences do not affect the application development process. Refer to *DB2 for Windows NT Application Programming* for more information.

Both ODBC and DB2 CLI allow multiple connections. Therefore, it is necessary to indicate which database the persistent object uses when it accesses the database table. The generated parts provide features to identify the database used for objects of `T`, `TManagerBase`, and derived classes like `TManager` and `TManagerTemplate`.

- ❑ `T::defaultDatastore` a class attribute, specifies the default datastore used for objects of `T` and `TManagerBase`.
- ❑ `T::objectsDatastore` specifies the datastore used for the object.
- ❑ `T::currentDatastore` retrieves the currently used datastore, which is `objectsDatastore`, if set, or `T::defaultDatastore` otherwise.
- ❑ `TManagerBase::objectsDatastore` specifies the datastore used for the object.
- ❑ `TManagerBase::currentDatastore` retrieves the currently used datastore for the object, which is `objectsDatastore`, if set, or `T::defaultDatastore` otherwise.
- ❑ `TD datastore::setAsDefaultDatastore` sets `T::defaultDatastore` to this datastore object.

## Impacts on the Application

You now know enough about the differences of embedded SQL and ODBC data access methods and how they affect the data access classes, to rebuild the application with the ODBC approach. It is wise to evaluate the impact of such a project change before you really begin the project.

You should keep the same names for files and working directories as used before. As you still access the same database, the names of tables, views, and columns have not changed, neither have the generated class and attribute names. But there are still several differences that might affect the project:

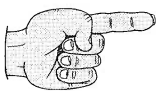
- ☐ The DB2 precompile step is no longer needed.
- ☐ The interface of the generated data access classes has changed.
- ☐ The datastore classes are derived from `IDatastoreODBC` instead of `IDatastore`.
- ☐ The database connection must be explicitly identified to the persistent objects, before they can access the tables.

The advantage of DB2 CLI and ODBC data access method over embedded SQL is that they do not require any precompile step when the application is built. When you create the Dacslib project, you have to select DB2 Precompile as one of the actions to build the project. The DB2 Precompile processes the .sqx files into .cxx files which then run through the compiler. Now, the MakeMake tool would complain that there are no files to be processed by the DB2 Precompile action.

The code generated from within the VisualBuilder is based on the interface definition, as provided in the .vbe files generated by the Data Access Builder. All names are still the same, and you have exploited only features common to all data access methods. As long as you do not exploit any specific features of the new ODBC data access classes, you need not regenerate any Visual Builder parts.

When you implement the logon function, you use an `IDatastore*` part to connect to the database (see Table 60 on page 338). At first sight, you may intend to replace this part with an `IDatastoreODBC*` part. But this would cause you to change the implementation, whenever you change the data access method.

#### Read This



Try to keep your code as portable and flexible as possible, so that required changes involve the least work.

As a second approach, you may choose one of the generated datastore classes, such as `PropertyDatastore`. The conceptual model of `PropertyDatastore` is a database that contains only a `Property` table. It does not know about the other tables.

The third approach is to create a new datastore class for the specific access to the REAL database. You derive this class conditionally from `IDatastore` or `IDatastoreODBC`. There is no symbol defined that identifies the type of data access method used for the generation. However,

the header files for these classes define a symbol to protect them from multiple inclusion. You can also use the symbols `_IDSMCOD_HPP_` and `_IDSMCON_HPP_` for conditional compilation.

No matter from which class you instantiate the datastore part, you need to set it as the default or object's datastore for all persistent objects and manager parts throughout the whole application. Again there are several implementation options.

Using the `IDatastoreODBC`, you can connect it to the `objectsDatastore` attribute of each persistent object and persistent object's manager. We do not recommend this implementation: it restricts the flexibility of choosing the data access method, it affects many parts, and thus it is hard to handle.

More appropriate, you can add a part of each persistent object to the `RealMain` view with the sole purpose of connecting the datastore part to the `defaultDatastore` attribute of them. This class attribute will affect the default for all persistence objects and managers instantiated thereafter. This approach is feasible for a small database and when the data access method never changes.

We choose an implementation with a `PropertyDatastore` part and custom logic to set the `defaultDatastore`. We implement the custom logic conditionally testing for the `_IDSMCOD_HPP_` symbol. Thus, we reserve flexibility and still keep the process simple for this application.

#### Code to Set the Default Datastore

```
#ifdef _IDSMCOD_HPP_
List_area::setDefaultDatastore(source->iRealDatastore);
Marketing_info::setDefaultDatastore(source->iRealDatastore);
Multidoc::setDefaultDatastore(source->iRealDatastore);
Property::setDefaultDatastore(source->iRealDatastore);
Property_log::setDefaultDatastore(source->iRealDatastore);
Property_address::setDefaultDatastore(source->iRealDatastore);
Prop_ad_log::setDefaultDatastore(source->iRealDatastore);
#endif
```

**Figure 148.** Default Datastore Setting Code

In a real project, you are likely to select the last implementation: Derive your own `REAL` datastore class and set the `defaultDatastore` of all tables within that datastore's constructor.

## Building the Data Access Classes

We recommend that you save the existing `Dacslib` project before you continue. Use the File Manager or Windows Explorer to rename the `D:\VR\Sources\Dacslib` directory to `D:\VR\Sources\DacsDB 2`. Create a new directory `D:\VR\Sources\Dacslib` to store the new generated files.

Open the Dacslib project. As the directory settings point to the newly created Dacslib directories, the Workframe project is empty. To generate the classes you proceed with the same steps as with using embedded SQL (see Chapter 6, “Mapping Relational Tables Using Data Access Builder,” on page 131), except that this time you explicitly specify the ODBC data access method:

1. Start the Data Access Builder from the Project pull-down menu.
2. Select **Create Classes** in the Startup window.
3. Select **ODBC Data Sources** as database type.
4. Select the database **REAL**.
5. As table filters use owner **USERID** and type **TABLE,VIEW** and click on **Connect**.
6. Generate the classes exactly as you did with embedded SQL data access method. Note that this time a .cpp file is generated for each class instead of a .sqx file.

Remove the action from the build settings, build the Dacslib project, and move all created parts in the VRDacs.vbb file:

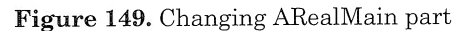
1. Select **Build normal** from the **Options** menu.
2. Deselect **DB2 Precompile** in the action list and click on **OK**.
3. Select **Rebuild all** from the **Options** menu.
4. Deselect **DB2 Precompile** in the action list and click on **OK**.
5. Click on the **Build normal** button in the toolbar.
6. Start **VisualBuilder**, import all .vbe files and move all generated data access parts to VRDacs.vbb.

## Rebuilding the Whole Application

Use PropertyDatastore instead of IDatastore throughout the property subsystem:

1. Open the **VR Property** project.
2. Double-click on the vrprop.vbb file to start Visual Builder.
3. Open the composition editor for the APropertyManagementView-part.
4. Click the right mouse button upon the RealDatastore variable.
5. Select the **Change type** menu item.
6. Enter **PropertyDatastore\*** in the entry field and select **OK**.

- Use `PropertyDatastore` in the `ARRealMain` part and add the custom logic (see Figure 149). Follow the steps in Table 68:



Step	Action
1	Start Visual Builder from the VReality project (select <b>Project</b> → <b>Visual</b> in the menu bar).
2	Select the loaded part file <code>vrmain.vbb</code> and open the composition editor of the visual part <code>ARealMain</code> .



<b>Table 68.</b> (Part 2 of 2) Changing ARealMain for ODBC Data Access Method	
Step	Action
3	Add a PropertyDatastore* part, <b>A</b> , on the free-form surface.
4	Reconnect all connections from RealDatastore to PropertyDatastore1: <ul style="list-style-type: none"> <li><input type="checkbox"/> Select the connection.</li> <li><input type="checkbox"/> Select the square that marks the end of the connection near RealDatastore.</li> <li><input type="checkbox"/> While holding the left mouse button drag this end to PropertyDatastore1 and drop it.</li> </ul>
5	Delete the RealDatastore part.
6	Rename PropertyDatastore1 to RealDatastore.
7	Add a new connection <i>ready</i> → <i>customLogic</i> from the free-form surface to RealDatastore and enter the code to set the default datastore for all persistent objects (see Figure 148).
8	Switch to the class editor and add the list of required include files: <pre> "ListArea.hpp" _List_area_HPP_ "MarkInfo.hpp" _Marketing_info_HPP_ "Multidoc.hpp" _Multidoc_HPP_ "Property.hpp" _Property_HPP_ "PropAdd.hpp" _Property_address_HPP_ "PropLog.hpp" _Property_log_HPP_ "PropAdLg.hpp" _Property_ad_log_HPP_ </pre>
9	Save and generate the part.
10	Close the editor and the Visual Builder.

From the smart option, select **Build any subprojects first** and then select **build** to incrementally rebuild the affected parts of the whole application.

Now that you have implemented the application independent of the data access method, you can easily switch: rename the Dacslib directory to DacsODBC, rename the DacsDB2 to Dacslib, use a tool like *touch* to set the timestamp for all header files in the Dacslib directory, reselect the DB2 Precompile option of the Dacslib project and build it, then incrementally build the whole application.

## Implementing the Database as dBase Files

With an ODBC version of the application you are ready to switch the underlying database engine with relatively small effort. In this section, you learn how to access dBase files (dbf) with ODBC and about the major differences between a DB2 database and a database in terms of dbf files. You implement the database as dbf files and add additional code to the application.

VisualAge for C++ for Windows comes with two ODBC driver packages. The INTERSOLV DataDirect ODBC Drivers for Windows 95 and Windows NT include various 32-bit ODBC drivers for both relational and flat-file database systems, as you can see in the following list:

- ☐ Oracle 7 ODBC Driver
- ☐ Sybase 10 ODBC Driver
- ☐ Text ODBC Driver
- ☐ Excel 4 ODBC Driver
- ☐ Excel 5 Workbook ODBC Driver
- ☐ DBase (2 and up) ODBC Driver
- ☐ SQL Server ODBC Driver
- ☐ Informix 5 ODBC Driver
- ☐ Ingress 6.4/04 Driver
- ☐ SQLBase ODBC Driver

The included INTERSOLV DataDirect ODBC Drivers for Windows 3.1 provides the 16-bit versions. The shipped dBase ODBC Driver supports various formats of dbf and index files. If you start the ODBC Administrator from the Control Panel, you see the 32-bit drivers in the driver list and the daxsamp data source. Daxsamp is a dBase database and comes with the Data Access Samples.

### A Small Introduction to DBF Files

Ashton Tate introduced dBase for IBM PC and compatibles in the 1980s. dBase defined a programming language, now often referred to as *Xbase*, and the dbf file format for data storage. The product consisted of an interpreter and the database engine that managed the dbf files.

In the last 10 years, a growing number of products such as Clipper, FoxPro, and Flagship supported the Xbase language. Further database products such as Lotus Approach and Microsoft Access use dbf files as a primary or alternative way to store data.

In relational database terms, you can regard a dbf file as the physical representation of a table. A header part contains the table definition and the file structure. The table rows are stored as records, where the

columns are mapped to fields in the record. dBase also supports indexes to increase the performance of accessing rows. It stores each index in a separate .ndx file.

In dBase III, memo files with the extension .dbt added the capability of storing variable text fields with more than 254 characters. And dBase IV provided a new file format mdx to store multiple indexes. Today you find several file formats to index dbf files, like the adx files in Lotus Approach.

The following table provides you with the list of field types in dBase IV dbf files:

<b>Table 69.</b> Data Types of dBase IV DBF Files			
<b>Name</b>	<b>Field Type</b>	<b>Bytes</b>	<b>Values</b>
Logical	L	1	T
Memo	M	2 147 483 647	“Could be very long”
Char(Length)	C	1 - 254	“Typical string”
Numeric	N	1 - 20	-10.20
Float	F	1 - 20	299.9544277
Date	D	8	19960310

Several features known in typical relational database systems like SQL, views, locking mechanisms were added to Xbase products. Some of them, such as locking mechanisms, led to additions and modifications to the simple and straightforward approach of dbf and index files. Other features, especially the concept of views, were not reflected in the file formats. You use the dBase IV file formats (dbf, mdx, dbt) to create the REAL database, as they are widely supported.

## Representing Databases with DBF Files

Databases are usually given an architecture using three layers of structures: external, logical, and internal. The external structure describes the views of different users to the database objects. The logical structure defines a plain, uniform, data model of the objects. The internal structure defines the physical representation of the objects in storage.

This architecture makes physical and logical independence possible. The physical independence allows changes (optimizations) in the physical representation without affecting the logical representation. Logical independence allows to add or modify different views to the database objects and still keep the same logical representation. A DBMS should provide the transformation between these layers transparently.

Obviously, the physical representation of a database could be implemented by a number of these file types (.dbf, .ndx, .mdx, .dbt). By convention, you keep all the files that represent a database together in one directory. In contrast to typical DBMS like DB2, the physical representation, the various file formats, is publicly known. This simple fact implies some important advantages and disadvantages:

- ☐ Multiple DBMS can access the same database. For example, you can build and manage a database with Lotus Approach, and access it from an application with an InterSolv ODBC Driver.
- ☐ Over time, these file formats constitute a common method to exchange databases. They provide much more functionality than simple text files and are supported by many products.
- ☐ The data integrity cannot be guaranteed by a DBMS, if the files are manipulated directly or with another DBMS.
- ☐ The physical data independence is not guaranteed. The files could be directly accessed and manipulated without any DBMS.
- ☐ The physical representation is fixed and the DBMS cannot optimize it for specific application and database setups. Of course, the DBMS can extend the physical representation to other files. However, other DBMS have no knowledge of these files and accessing the database with multiple DBMS could lead to inconsistencies.
- ☐ New database features cannot be added to the existing file formats. That is why there are so many different file formats.

Unlike the use of DB2, you face a number of restrictions inherent to dBase IV file format. Most of these restrictions are also true for the other existing file formats:

- ☐ Table names are limited to eight characters.

As a result of mapping tables to files and the 8.3 convention of DOS file names (8 characters for the base name, 3 characters for the extension), the restriction to eight characters applies to table names.

- ☐ Owner qualifiers for table names are not supported.
- ☐ Attribute names are limited to ten characters.
- ☐ Some SQL data types are not explicitly provided.

A DBMS could map the unsupported data types (SMALLINT, INTEGER, TIMESTAMP) to the existing ones.

- ❑ Primary keys are not explicitly stored.

You can specify indices in the index files as unique and use them as keys. But you cannot indicate primary or foreign keys of tables explicitly.

- ❑ There is no representation of NULL in any field type.

*“Null is a special value that is used to represent ‘value unknown’ or ‘value inapplicable’” (An Introduction to Database Systems; Volume I, C.J. Date, Addison Wesley, 1986).* You need this special value in many situations and it is included in the SQL standard.

For example, SQL provides the NOT NULL option for columns in a table definition. You should use this option for primary keys to avoid possible inconsistencies in the database. We also recommend using it for required fields.

A sophisticated DBMS may circumvent some of these limitations, so that you could disregard them for the logical representation of the database. To access the REAL database, use the DBase ODBC Driver shipped with VisualAge for C++ for Windows. You do not have to buy or install another ODBC driver.

This driver provides a rather straightforward mapping between the physical and the logical representation. All restrictions mentioned above also apply to the logical representation. In conjunction with the dBase IV file formats, the driver supports the data types shown in Table 70:

<b>Table 70.</b> Data Types of ODBC dBase Driver		
<b>Name</b>	<b>Precision</b>	<b>Example</b>
Logical	1	1
Memo	2 147 483 647	“Could be very long”
Char(Length)	254	“Typical string”
Numeric(Precision, Scale)	19	-10.20
Float(Precision, Scale)	19	299.9544277
Date	10	{10-03-1996}
<b>Note:</b> All data types are nullable.		

Because of the lack of a special NULL representation, the value NULL is mapped to an existing value of the data type and all data types are defined as nullable. The driver does not provide the layer of an external representation. You can create tables, but not views!

## Impacts on the Application

Knowing the basic concepts and restrictions of dbf files you are prepared to implement the application with dbf files. But first, let us list the impacts to the existing database and application.

### ❑ **Table and column names are limited.**

The REAL database has several tables and columns with longer names, so the database definition changes. In the Data Access Builder, you map these shortened table and column names to data access classes and attributes with the same long names as used with the DB2 database. The generated classes still provide the same interface to the remaining application.

### ❑ **Owner qualifiers for table names are not supported.**

This affects only the database definition and the generated source code. The interface of the data access classes does not change.

### ❑ **Some SQL data types are not explicitly provided.**

You can directly map VARCHAR to Memo and DECIMAL to Numeric. The two unsupported data types in the definition of the REAL database are SMALLINT and TIMESTAMP. You replace SMALLINT by Numeric and TIMESTAMP by Char(25).

Take a look at how the Data Access Builder maps the data types to C++ types. While SMALLINT is mapped to short and TIMESTAMP to char\*, Numeric is mapped to double and Char to IString&. These differences are also reflected in the class interfaces and impact the whole application.

With the Visual Builder, you must reimport the part interfaces and regenerate the parts that use the data access classes. This way you ensure that the differences are reflected properly.

If you intend to enable your application for DB2 and dbf files at the same time in a real project, you should anticipate these differences, and restrictively use the data types also available in dbf files.

### ❑ **Primary keys are not supported.**

A primary key is defined to uniquely identify a row in a table. To ascertain this, you create a unique index for the columns used as primary keys. Whenever you open the class settings of a table stored in a dbf file, the Data Access Builder displays a warning that no primary key is defined. By default, the first column is used

as a data identifier. In all tables of the REAL database the first column represents the primary key and you do not need to change the default settings.

❑ **There is no representation of NULL.**

NULL values play a fundamental role for assumptions about relational databases, most important about the database integrity. Your application code compiles and link without changes. But it is your responsibility to analyze the impact on your business logic.

For example, a SQL query such as `SELECT * FROM Buyer WHERE income = 0` may produce different results with standard SQL products like DB2 and with dbf files. The column income is defined nullable in both database definitions. DB2 delivers all rows that have an existing income value of 0. With dbf files you also get rows that have an undefined income, as the NULL value is represented by 0.

❑ **Views are not supported.**

The REAL database definition includes three views. Two of them, `Buyer_info` and `Prop_ad_log`, provide a join to simultaneously show columns of different tables in the same container. The third, `List_area`, uses a distinctive selection and returns a list of all areas, in which properties are available.

You can minimize the impact on the remaining application, if you implement this functionality in the same way as before. You create three classes with the same class interface as those generated by the Data Access Builder for the DB2 views.

In summary, you perform the following steps to implement the application with dbf files:

1. Create and register the REAL database as dbf files.
2. Map the tables to data access classes.
3. Implement classes that behave like database views.
4. Rebuild the application.

## Creating the DBF Files

You can create the dbf files with Lotus Approach or any other database product that stores data in dbf files. However, Lotus Approach has its own index file format (adx). While it can manage existing dBase IV files simultaneously, it does not allow you to create them. Appendix C, "Database Definition," on page 515 contains the SQL commands to create the dbf files. Use them as a reference when you create the dbf files and data fields.

If you have no database product at hand or you are not interested in these details, you can use the existing dbf files of the full application. The database directory is located in \VR\Sources\Misc\DBF. Skip the remainder of this section and continue with registering these files as ODBC data source.

As an alternative you build the database by issuing SQL data definition commands against an empty ODBC data source. You need a tool that provides you with an environment to access ODBC data sources and to perform SQL commands. The sample ODBC Admin Demo in the ODBC Software Development Kit (SDK) is sufficient. This SDK is included in the Microsoft Developer Network Professional (Level 2). In the following, we assume that you have installed the ODBC SDK including samples and Win32 (32 bits Windows API) components.

The license agreement for the InterSolv ODBC driver package included in VisualAge for C++ for Windows does not allow the use of these drivers for creation of databases. If you try the following steps with the InterSolv ODBC dBase Driver, the driver informs you about the license agreement and that you have to purchase a full version. Therefore, we describe the use of the Microsoft ODBC dBase Driver.

In the CD-ROM that accompanies this book, there are six files with an extension of .sql that contain SQL commands to set up the REAL database. Compare the creatab.sql with the corresponding .ddl file, the differences reflect the mentioned restrictions of dbf files. All table and column names apply to the limits of the dbf format. The owner qualifier, the view definitions, the not-null option and the *primary key* option are deleted. The data types SMALLINT, VARCHAR, TIMESAMP, DECIMAL are replaced by the best fitting dbf equivalents.

- ☐ creatab.sql creates the dbf files for the tables with Microsoft's driver.
- ☐ creatabi.sql creates the dbf files for the tables with InterSolv's driver.
- ☐ creavw.sql creates dummy dbf files for the views with Microsoft's driver.
- ☐ creavwi.sql creates dummy dbf files for the views with InterSolv's driver.
- ☐ index.sql creates the mdx files for the tables.
- ☐ update.sql fills the dbf files with the initial data.

While the ODBC specification enables accessing different databases dynamically, it does not ensure compatibility. Even though both ODBC drivers access the same physical database, they have a different syntax for data type definitions and data constants! While Data Access Builder works fine together with the InterSolv driver, you cannot use it in combination with the Microsoft driver.



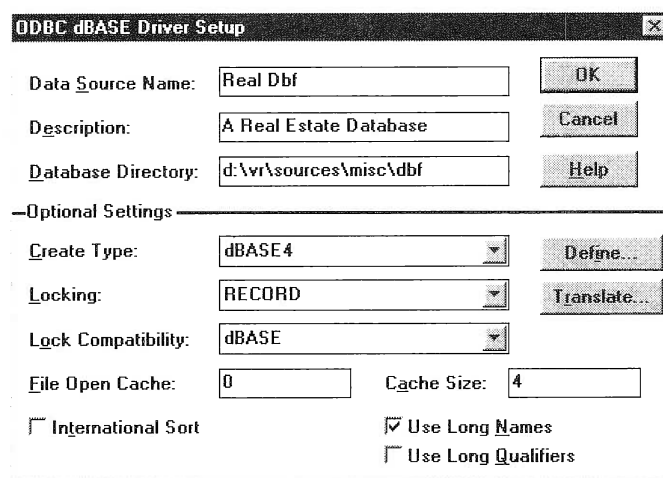
With the following steps, you build the database:

1. Create an empty directory where all database files will be located.
2. Add a new ODBC data source in the **32-bit ODBC** administrator.  
Using the **Microsoft dBase Driver**, provide a data source name such as **REAL Build** and specify the database directory.  
If you cannot specify a directory in the file dialog provided in Windows 95, you must patch the directory information in the registry with the regedit tool. The entry DefaultDir is in Desktop\HKEY\_USERS\.Default\Software\ODBC\ODBC.INI\REAL (in Windows 95, Desktop is replaced by *My Computer*).
3. Start the sample **ODBC Admin Demo** in the **ODBC SDK** folder.
4. Select the **REAL Build** data source.
5. Select the **File→Execute** menu item to get the Execute File window.
6. Specify the semicolon character as terminator, a maximum length of 9999 and the creatab.sql as file name and close to execute these SQL commands by clicking on the **OK** button.
7. Execute the index.sql.
8. Execute the update.sql.
9. Close the **ODBC Admin Demo** sample.

## Registering the ODBC Data Source

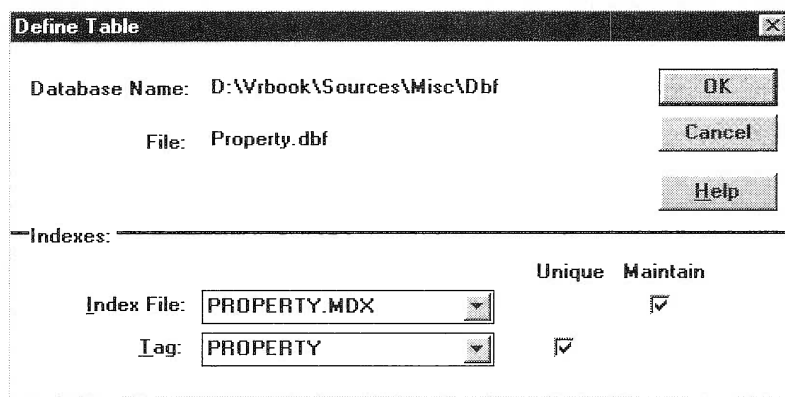
You now register an ODBC data source for the database together with the InterSolv ODBC dBase Driver:

1. Open the **Control Panel** window.
2. Open the **32-bit ODBC** administrator window.
3. To rename an existing REAL data source, select it, and click on **Setup**, change the **data source name** and click on **OK** again.
4. Click on **Add** to open the **Add Data Source** window.
5. Select the **InterSolv 2.11 32-bit dBASEFile (\*.dbf)** and click on **OK**.
6. Fill in the information in the **ODBC dBase Driver Setup** window (Figure 150). Enter **REAL** as data source name, provide a meaningful description and type in the directory that contains the dbf files. Select the **dBASE4** create type.



**Figure 150.** Registering dbf Files as an ODBC Data Source

7. Click on **Define** to open a standard-file open window named **Define File**.
8. In the **Define File** window (Figure 151) select the first dbf file and click on **Open** to define the special indexes using the **Define Table** window. Select in the **Index file** drop-down list box the mdx file with the same name as the dbf file. Select the **Maintain** check box to associate this index file with the dbf file. In the **Tag** drop-down list box, select the tag and verify that the **Unique** check box is selected. Click the **OK** button to save this information.



**Figure 151.** Associating mdx and dbf Files

9. Perform the previous step for all remaining dbf files in the directory. Then, click on **Cancel** to return to the **ODBC dBASE Driver Setup** window
10. Click on **OK** to save the new ODBC data source REAL.
11. Close the **ODBC Administrator** and the **Control Panel** windows.

## Generating the Data Access Classes

We recommend that you save the existing Dacslib project again, before you continue. Use the File Manager or Windows Explorer to rename the D:\VR\Sources\Dacslib directory to D:\VR\Sources\DacsODBC. Create a new directory D:\VR\Dacslib to store the new generated files.

Open the Dacslib project. As the directory settings point to the new created Dacslib directories, the Workframe project is empty. You generate the classes as you did for the DB2 database with ODBC (See “Building the Data Access Classes” on page 393.).

The Data Access Builder allows you to change the class and attribute names used in the generated code. With this feature you can keep the interface of the Dacslib project as much as possible. You change the class and attribute names in the settings window of the class icon.

1. Select the **Data Access Builder** from the Project pull-down menu.
2. Select **Create Classes** in the Startup window.
3. Select **ODBC Data Sources** as database type.
4. Select the database **REAL**.
5. Click on **Connect**, then on **Select All** and **Create Classes**.
6. Click on **OK** in the **Create Class - Class Options** window to accept the generation of C++ Visual Builder parts.
7. Double-click on the Marketing\_info class icon to get the settings window. The Data Access Builder warns you that no primary key is defined for this table. The dbf files do not explicitly support primary keys.
8. Change the class name to MarkInfo in the Names setting page. Change the attributes names to their shortened equivalents: prop\_id, days, commission and down\_pay in the attributes page. Close the settings window.

Repeat this step for all tables that have shortened names, as listed below in Table 71.

9. Generate the code for all classes.

10. Close the Data Access Builder and save the information in the file `real.dax` in the `\VR\Sources\Dacslib` directory.

**Table 71.** Shortened Table and Attribute Names of REAL dbf Files

DB2 Name	dbf Files
Buyer marital_status	Buyer marital
Buyer_address	BuyAdd
Buyer_log creation_timestamp last_update	BuyLog creation update
Marketing_info property_id days_on_market commission_rate down_payment_rate	MarkInfo prop_id days commission down_pay
Multidoc multidoc_id	Multidoc doc_id
Preference	Prefer
Property property_id description	Property prop_id descript
Property_address	PropAdd
Property_log property_id download_timestamp last_update	PropLog prop_id download update
Sale_transaction transaction_id last_update agreement_form_id property_id	Sale trans_id update form_id prop_id

Notice that we list in Table 71 all the tables and views that are part of the complete application.

## Creating Classes for Database Views

Dbf files do not provide views, but the existing application relies on them. Instead of changing the existing logic, you provide the class code for the views yourself. This may sound tedious, but with the help of the Data Access Builder, you achieve this without much effort.

A database view is very similar to a database table. It behaves exactly like a table. The major difference is that there is no physical representation of the view contents, but the contents is built of the contents of one or multiple tables.

Comparing the generated .vbe files of the Multidoc table and the List\_area view (these are the smallest.), you notice only one difference in the class interfaces. Only the table provides actions add, update, delete to modify the table contents. Compare also the generated source code .cpp in the \VR\Sources\DacsODBC subdirectory. Of course, the view implementation misses the methods that implement the add, update, and delete actions. The only major difference lies in the SQL select statement of the methods for select and retrieve rows of the table. The table select accesses the columns of the table, the view select joins existing tables and provides a subset of the table columns as columns of the view.

You can utilize this inherent similarity of tables and views. Define tables with the same column definitions as the views you want to simulate. Compare the table definitions in Figure 152 with the view definitions in Table 77 on page 516. Then, generate the data access classes for these dummy tables with the Data Access Builder. Finally, change the two select statements for selection and retrieval, so that they reflect the view definition.

```

create table BuyInf
(buyer_id char(11), first_name char(20),
 last_name char(20), income numeric(7,0),
 work_phone char(12), home_phone char(12),
 street char(40), area char(40),
 city char(40), state char(2),
 zip_code char(5), max_price numeric(8,0),
 min_price numeric(8,0), max_size numeric(5,0),
 min_size numeric(5,0), bedrooms numeric(2,0),
 bathrooms numeric(2,0), stories numeric(2,0),
 heating char(30), cooling char(30));

create table PropAdLg
(prop_id char(5), size numeric(5,0),
 bedrooms numeric(2,0), bathrooms numeric(2,0),
 area char(40), city char(40),
 state char(2), status char(15),
 price numeric(8,0), commission numeric(4,2),
 down_pay numeric(4,2));

create table ListArea
(area char(40));

```

**Figure 152.** Data Definition for Dummy Tables

The steps are these:

1. Create an empty directory where the dbf files will be located.
2. Add a new ODBC data source in the **32-bit ODBC** administrator.  
Using the **Microsoft dBase Driver**, provide a data source name such as **REAL Build Views** and specify the database directory.
3. Start the sample **ODBC Admin Demo** in the **ODBC SDK** folder.
4. Select the **REAL Build Views** data source.
5. Select the **File→Execute** menu item to get the Execute File window.

Specify the semicolon character as terminator, a maximum length of 9999 and the **creavw.sql** as file name and close to execute these SQL commands by clicking on the **OK** button. Close the **ODBC Admin Demo** sample.

6. Add a new ODBC data source in the **32-bit ODBC** administrator.  
Using the **InterSolv 2.11 32-bit dBASEFile (\*.dbf)**, provide a data source name such as **REAL Views**, specify the database directory and select the **dBASE4** create type and click on **OK**. Close the **32-bit ODBC** administrator.
7. Generate the classes for the views with the Data Access Builder

In the settings page for each class, select the **Read-Only** access check-box (the Read-Only check-box allows you to specify that all instances of your class not be permitted to write, update, or delete your database information), change the shortened table and attribute names listed in Table 72 and generate the source files.

<b>Table 72.</b> Shortened Table and Attribute Names of REAL View dbf Files	
<b>DB2 Name</b>	<b>dbf Files</b>
Buyer_info	BuyInf
List_area	Listarea
Property_ad_log property_id commission_rate down_payment_rate	PropALog prop_id commission down_pay

8. Edit the three source files, BuyInf.cpp, ListArea.cpp and PropALog.cpp and search for and modify the two SQL select statements as shown in Figure 153. Replace the WHERE by an AND in the select method, where the select statement is called.

```

...
Prop_ad_logRetrieveStatement(IDatastoreODBC* aDatastore
                           , Prop_ad_log& aProp_ad_log)
    : IDSStatementRetrieve(
        "SELECT "
        "SIZE" ", "
    ...
        "DOWN_PAY"
        " FROM PROPERTY A, PROPADD B, "
        "PROPLUG C, MARKINFO D WHERE "
        "A.PROP_ID = ADDRESS_ID AND "
        "A.PROP_ID = C.PROP_ID AND "
        "A.PROP_ID = D.PROP_ID AND "
        "A.PROP_ID = ?"
        , aDatastore )
...
const IString Prop_ad_logSelectStatement::selectStatement =
    "SELECT "
    " A.PROP_ID " ", "
    "SIZE" ", "
...
    "DOWN_PAY"
    " FROM PROPERTY A, PROPADD B, "
    "PROPLUG C, MARKINFO D WHERE "
    "A.PROP_ID = ADDRESS_ID AND "
    "A.PROP_ID = C.PROP_ID AND "
    "A.PROP_ID = D.PROP_ID";
...
Prop_ad_logManagerBase& Prop_ad_logManagerBase::select(const char* clause)
{
    try {
        Prop_ad_logSelectStatement aStatement(currentDatastore(),
        IString( " AND " ) + clause );
    ...

```

**Figure 153.** Changed Select Statements in PropAdLg.cpp

Compare these statements with the view definitions in Appendix C, “Database Definition,” on page 515 and modify the other select statements appropriately.

10. Select **Build normal** from the **Options** menu, deselect **DB2 Pre-compile** in the action list, and click on **OK**.
11. Select **Rebuild all** from the **Options** menu, deselect **DB2 Pre-compile** in the action list, and click on **OK**.
12. Click on the **Build normal** button in the toolbar.
13. Start **VisualBuilder**, import all .vbe files and move all generated data access parts to VRDacs.vbb.



**Read this**

Because of a bug in `ipoattr.hpp` you cannot update a string attribute of a persistence object with an empty string at the first time in combination with the InterSolv ODBC dBase driver.

If you have not installed any fix, you should change the constructor of the string attribute in `ipoattr.hpp`:

```
class IPOStringAttribute : public IPOAttributeBase
{
protected:
    IString theString;
public:
    IPOStringAttribute(): IPOAttributeBase(true)
                        , theString("") {}
}
```

## Building the Application

With the previous steps, you achieved minimal impact on the remaining parts of the application. Only the changes of the data types are visible in the interface. If you expect to deliver both access methods, restrict your changes to the common data types.

To reflect the interface changes in all parts of the application, regenerate all VisualBuilder parts that depend on the data access classes. In the Visual Realty application, the changed data types do not affect the use of the data access classes by the other VisualBuilder parts. Therefore, it is sufficient to just rebuild the whole application.

Set the *smart option* rebuild subparts first and then select the **build** action to incrementally rebuild only the affected parts of the whole application. The Dacslib project should be built separately for the embedded SQL data access method. In this case, the build settings differ from those of the Visual Realty project and are not correctly built as a subproject of Visual Realty.

Keeping the application independent of data access method and DBMS, makes it possible to switch easily between the three sub-projects DacsODBC, DacsDB2, and DacsDbf.

## Enhancing the Data Access in the Application

In general, the generated data access classes provide the basic features to work with data sources. In the following section, we give you examples of more advanced features and outline how to enhance the classes in your application.

When you switch to the ODBC access method, you learned the benefit of a derived datastore class that represents the whole database. In “Creating Classes for Database Views” on page 408 you modified the generated code to implement views and join multiple tables.

Typically, you derive new classes to add specific features. In certain circumstances this is not possible and the only way is to change the generated data access classes. We recommend providing a script or a batch file that automatically modifies the generated code when you build the project.

In future releases, the Data Access Builder will offer more features and greater flexibility.

### Ordering Data

You have two possibilities to order your data. You can either order the data whenever it is displayed to the user, or you can store it in a defined order in memory.

To order data in a container the `IContainerControl` provides two methods for sorting: `sortByIconText` and `sort`. With the first method, you sort the objects in the container using their icon text. Use the `sort` method together with a comparison function that you implement for any application-specific sorting.

Perform the following steps to provide this sorting function:

- ❑ Derive a class from `IContainerControl::CompareFn`.
- ❑ Override the abstract method `isEqual` in your derived class. `isEqual` returns an integer value that indicates whether the first object is less than, equal to or greater than the second.
- ❑ Create an object of your derived comparison class
- ❑ Use this object as a parameter when you call the `sort` method

You can use the `select` method of `TManager` to order the data in memory. You simply include a `order by` definition in the `select` clause. After elements are added or updated, you have to call the `select` method again.

## Buffering Data

The `TManagerBase` class provides the refresh and select methods to retrieve multiple rows. The refresh method retrieves all rows of a table. You might prefer a buffered retrieve method for huge tables. Typically, you implement this with a cursor. However, the Data Access Builder does not yet support cursors in this release. You must modify the generated code to add buffered refresh using a cursor.

If you connect the items attributes of a huge `TManager` and a container, as done in the `APropertySearchResultView`, performance decreases. If you are concerned about the performance issue rather than allocated memory, you may implement a buffered sequence class to connect to the container. The buffered sequence would contain only a subset of the `TManager` elements.

In addition, you need to implement a scroll handler for the container to constantly update the buffered sequence when scrolling. The implementation of a derived sequence class for buffering and a scroll handler is beyond the scope of this book.

## Displaying Data

In “Using Collection Combination-Box Control” on page 201, we describe how to create a string-generator class from the ground up and how we associate it with a collection combination-box control to tailor contents of that collection. In this section, we show you two other ways of tailoring the contents of a collection-list box control (the examples apply to any collection-list box or collection combination-box controls):

- ❑ Overriding the `asString` method of the part that is displayed in the collection-list box.
- ❑ Tailoring the behavior of an existing string-generator class.

### ***Overriding the `asString` Method***

Each object that you want to display in a collection-list box must be provided with an `asString` method to get its equivalent `IString` form. If an object is already provided with an `IString` method, you can override it with your own method by subclassing its corresponding class. The `asString` method needs only to return an `IString` object.

The only safe way to override the `asString` method of a Data Access Builder part is to derive a class from it. In effect, you should not modify the code generated by Data Access Builder because you would lose all of your changes if you generate the code from the same part later.

You use the `TManagerTemplate` class created by Data Access Builder when you map a relational table `T` to a `T` class. `TManagerTemplate` is used to define persistent objects derived from the mapped class.

In the following paragraphs, you modify the `TinyApp` application (see Section “Using Data Access Builder Parts with Visual Builder” on page 140) to use a new class, `MyProperty` class, derived from `Property`. This new class is provided with an `asString` method which overrides the virtual `asString` method of `Property`. You provide a copy constructor also since it is needed by any derived class in order to be used within the derived class manager.

To modify the `TinyApp` application, follow these instructions:

1. Open a command prompt session and change to the `D:\TEST` directory where your `TinyApp` code is located.
2. Start the Visual Builder from this directory and load `TINYAPP.VBB`, `D:\VR\Sources\Dacslib\DACS.VBB` and `D:\IBM-CPPW\IVB\VBDAx.VBB`.
3. Create a nonvisual part, `MyProperty`, which inherits from `Property` as follows:

Field	Value
Class name	<code>MyProperty</code>
Description	A Derived Property
File name	<code>TinyApp</code>
Part type	nonvisual part
Base class	<code>Property</code>

- Click on the **Open** push button. The Part Interface Editor is displayed.
4. Switch to the Action page and the following method:
    - A copy constructor:  
`MyProperty::MyProperty(const MyProperty& partCopy)`
    - An `asString` method with the following signature:  
`QString asString() const`
  5. Switch to the Class Editor and fill in the **User .hvp file** entry field with `MyPrprty.hpv` and the **User .cpv file** entry field with `MyPrprty.cpv`.
  6. Save the part and generate the part source and features source code.
  7. Edit the `MyPrprty.cpv` file and modify it as shown in Figure 154.

8. Open the TinyApp visual part and replace the PropertyManager part by a PropertyManagerTemplate part (add the part to the free-form surface by using the **Option** → **Add parts...** from the Composition Editor menu bar). The connections should remain the same.
9. Double-click on the **PropertyManagerTemplate** class and set its *itemType* attribute with **MyProperty**.
10. Double-click on the **collection view list box** and change the *Item type* to **MyProperty\***.
11. Save and generate the part source code and its make file.
12. Compile and link-edit your TinyApp application.

```
//
// Feature source code generation begins here...

MyProperty::MyProperty(const MyProperty& partCopy)
    :Property(partCopy)
{
}

IString MyProperty::asString() const
{
    return property_id();
}

// Feature source code generation ends here.}
//....
```

**Figure 154.** Overriding the asString Method

When you run the application, only the property identifiers are displayed in the collection list box since the `asString()` method returns the property identifier. You do not associate a string generator class with the collection view list box.

### ***Tailoring the Behavior of a StringGenerator Class***

TManagerTemplate class contains an embedded stringGenerator class that rules how the persistent objects are displayed within a collection list box. Each persistent object is provided with a `forDisplay` method which returns a string containing the concatenated `asString` representation of each of the attributes of the object that has been marked in the Data Access Builder to be returned by this method. The default separator ‘.’ is inserted between each attribute. The attributes to be displayed are marked by checking the **is Displayed** check box in the Attributes page of the object’s settings notebook (see Figure 47 on page 138). The `forDisplay` method is used by the embedded stringGenerator class to display the persistent objects in a collection list box. In

the next sample you create another derived class of `Property` and override its `forDisplay` method. Then you provide the collection view list box with the string generator of the `TManagerTemplate` class.

To modify the `TinyApp` application, follow these instructions:

1. Open a command prompt session and change to the `D:\TEST` directory where your `TinyApp` code is located.
2. Start the Visual Builder from this directory and load `TINYAPP.VBB`, `D:\VR\Sources\Dacslib\DACS.VBB` and `D:\IBM-CPPW\IVB\VBDA.VBB`.
3. Create a nonvisual part, **MyProperty2**, which inherits from `Property` as follows:

Field	Value
Class name	<code>MyProperty2</code>
Description	A Derived Property
File name	<code>TinyApp</code>
Part type	nonvisual part
Base class	<code>Property</code>

- Click on the **Open** push button. The Part Interface Editor is displayed.

4. Switch to the Action page and the following method:
  - A copy constructor:  
`MyProperty2::MyProperty2(const MyProperty2& partCopy)`
  - A `forDisplay` method with the following signature:  
`IString forDisplay(const char* sep=" ") const`
5. Switch to the Class Editor and fill in the **User .hvp file** entry field with `MyPrprt2.hvp` and the **User .cpv file** entry field with `MyPrprt2.cpv`.
6. Save the part and generate the part source and features source code.
7. Edit the `MyPrprt2.cpv` file and modify it as shown in Figure 155.
8. Open the `TinyApp` visual part and replace the `PropertyManager` part by a `PropertyManagerTemplate` part (add the part to the free-form surface by using the **Option** → **Add parts...** from the Composition Editor menu bar). The connections should remained the same.
9. Double-click on the **PropertyManagerTemplate** class and set its *itemType* attribute with **MyProperty2**.
10. Double-click on the **collection view list box** and change the *Item type* to **MyProperty2\***. In the **String generator** entry filed enter:  
`PropertyManagerTemplate<MyProperty2>::stringGenerator()`

11. Save and generate the part source code and its make file.
12. Compile and link-edit your TinyApp application.

```
//  
// Feature source code generation begins here...  
  
MyProperty2::MyProperty2(const MyProperty2& partCopy)  
    :Property(partCopy)  
{  
}  
  
IString MyProperty2::forDisplay(const char* sep) const  
{  
    return "*****" + property_id() + "*****";  
}  
  
// Feature source code generation ends here.  
//....
```

**Figure 155.** Overriding the forDisplay Method

When running the application, the forDisplay method is used by the stringGenerator class to display the property identifier of the MyProperty2 object in the collection view list box.

## Adding Calculated Attributes

Suppose you like to add a further column to the AProperty-SearchResultView container that shows the total price of a property. The Prop\_ad\_log class provides attributes size and price per square foot, and the total price is calculated as  $\text{total\_price} = \text{size} * \text{price}$ .

You can implement a derived class Property\_view of Prop\_ad\_log that adds a getTotal\_price and a total\_priceAsString method. You use this class as element type of the container. You also provide a manager class Property\_viewManager by instantiating the Prop\_ad\_logManagerTemplate class with Property\_view and connect the container items attribute to the items attribute of a Property\_viewManager object.

The code snippet for these two classes is given in Figure 156.

```

#ifndef _Propview_HPP_
#define _Property_view_HPP_

#ifndef _Propadlg_HPP_
#include "propadlg.hpp"
#endif

class Property_view : public Prop_ad_log
{
    const double total_price() const
    { return size() * price(); };
    IString total_priceAsString() { return IString( total_price()); }
}

class Property_viewManager :
    public Prop_ad_logManagerTemplate< Property_view> {};
#endif

```

**Figure 156.** Derived Property\_view class Propview.hpp

If you also want to add notification for this calculated attribute, you would override the `setPrice` and `setSize` methods to call the respective base versions and additionally notify the observers of the change of `total_price`.





# 12

## More about SOM...

The purpose of this chapter is to introduce the SOM and Direct-to-SOM (DTS) basics. We do not pretend to provide you with a comprehensive description of the SOM technology; many books have been written on the subject (refer to “Related Publications” on page xxx for more information).

After reading this chapter, you should be able to:

- ☐ Understand what SOM is, and how it complements C++.
- ☐ Write a simple class with the interface definition language (IDL), and use it from a C++ client code.
- ☐ Write and modify a SOM dynamic link library.
- ☐ Understand what DTS is, and how you can take advantage of it to reuse your existing C++ code while enjoying the beauties of SOM.
- ☐ Use a DTS class from the Visual Builder.

In this chapter, we assume that you are aware of the object-oriented programming principles. If you think your memory needs some refreshing, please read our “Object Talk” on page 6.

## The Need for SOM

A major goal of Object-Oriented Programming (OOP) is to write programs that can be more easily reused and extended than those written using conventional programming languages. OOP offers several advantages, such as encapsulation, which allows a clear separation between the definition of an object and its possible implementations. Encapsulation basically allows you to request a service from an object without caring about the way this service is going to be performed. With OOP, one developer would not have to redesign each application from scratch, but rather reuse a library of predefined components created during previous developments or provided by a software vendor.

Reality, however, is different, especially with the C++ language. A software vendor needs to provide a library not only according to the target platform, but often according to the target compiler. Similarly, a library needs to be totally rewritten if it is expected to be used in another programming language, such as Smalltalk. For the software vendor, managing the different versions of this library and redistributing new releases to all customers can soon become a nightmare.

SOM was developed to address the problems of OOP while preserving its benefits. The IDL allows any supported language to implement objects and their methods, while using a different language for the client application. Object libraries can be changed without requiring the client code to be recompiled. The Distributed SOM (DSOM) framework provides the foundation for distributing objects. SOM is the programming interface for the OS/2 Workplace Shell and OpenDoc.

In the early 1990s, the Object Management Group (OMG) set forth its Common Object Request Broker Architecture (CORBA) standards to provide interchangeability of objects. The version of SOM (2.1) delivered with the VisualAge for C++ product fully conforms to CORBA 1.1.

## SOM: A Complement to C++

If you wonder why you should use SOM rather than C++, then a possible answer is that you can use SOM as a complement to C++. SOM was not intended to replace this or that programming language, but rather to complement them in the following areas:

- ☐ Release-to-release binary compatibility, or the need to recompile client code each time you change a class library used by this code.
- ☐ Language neutrality, or the need to rewrite a library in order to call it from a specific programming language.

## Release-to-Release Binary Compatibility

Let us consider the following scenario: You are the proud author of a C++ library that implements a calculator class for adding numbers. Each of the 3000 developers working in your organization uses that library in its own code. Suppose that you want to greatly enhance your calculator class by adding a `subtract()` virtual method. You also want to store the result of any operation in an instance variable and provide accessors for querying the value of that result. You now face the task of updating the library and redeploying it on 3000 desktops. Moreover, the 3000 developers need to recompile their applications, even if they do not care about the features you have added to the library.

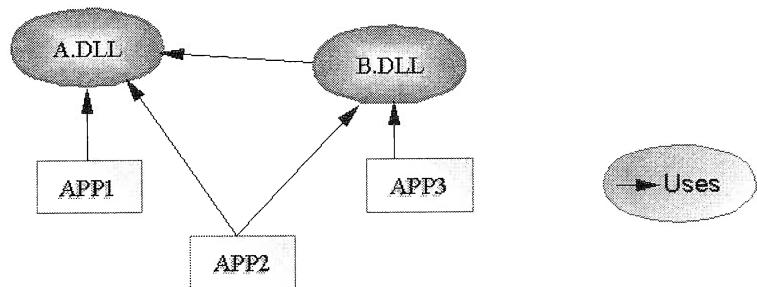
If your class library is implemented using SOM, then developers who do not need to subtract numbers will not even notice that you have updated the library. The SOM library continues to support existing applications without requiring any change. This feature is called release-to-release binary compatibility (RRBC).

### *Why is recompilation needed for C++ Libraries?*

In conventional C++, you can safely:

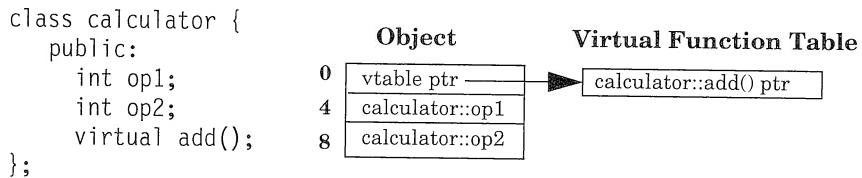
- ☐ Add nonvirtual member functions.
- ☐ Add static data members.
- ☐ Modify function bodies that are not defined inline.
- ☐ Add virtual functions to classes with no subclasses, or whose subclasses never add virtual functions.

Any other change requires the recompilation of all subclasses and client code. If you consider the situation depicted in Figure 157, any change in A.DLL requires B.DLL, APP2, and APP3 and all other applications that use it to be rebuilt and **simultaneously** reshipped.



**Figure 157.** Impact of Changes in a C++ Library

The rationale behind the need for recompilation rests in the way C++ libraries are built. Let us consider the following definition and its corresponding representation:



When you access the `op1` or `op2` instance variables, you do not access them through their name but through their offset—that is, their address in the library. Carelessly inserting an integer instance variable at the beginning of the `calculator` class definition would shift all offsets by 4, leading to unpredictable results at run time. Similarly, any class inheriting from the `calc` class would have to be recompiled if you add a new virtual function to the `calc` class.

SOM supports RRBC by maintaining a list of all methods introduced by a class, called the *release order* for the class. An IDL modifier is used to specify the release order for the class, listing each method in order by name, so that you can define new methods anywhere in the code while adding them at the end of the release order list.

## Language Neutrality

Let us now suppose that you want to share your library with Smalltalk programmers. Although the interface of the class is the same—that is, the class is providing exactly the same service to C++ or Smalltalk programmers—you need to rewrite the definition of your calculator class for the Smalltalk language.

SOM is language neutral, meaning that SOM class developers and users do not need to use the same programming language. The CORBA specification defines the IDL for describing class interfaces. You must use IDL to define the interface of the calculator class, then use the SOM compiler to generate the equivalent definition in a specific language such as C++. Thanks to the object-oriented encapsulation concept, a client application written in a different language such as Smalltalk or even C can use your SOM class.

You should now have a good idea of what you can use SOM for. We now introduce some important concepts of the SOM technology, such as the concepts of classes and metaclasses, as well as the SOM run time.

#### Warning



SOM is a powerful tool. However, you should be aware that a SOM-enabled class may run more slowly than the equivalent class implemented in pure C++.

Therefore, you must weigh the many benefits from SOM against the negative effect it may have on the global performance of your program.

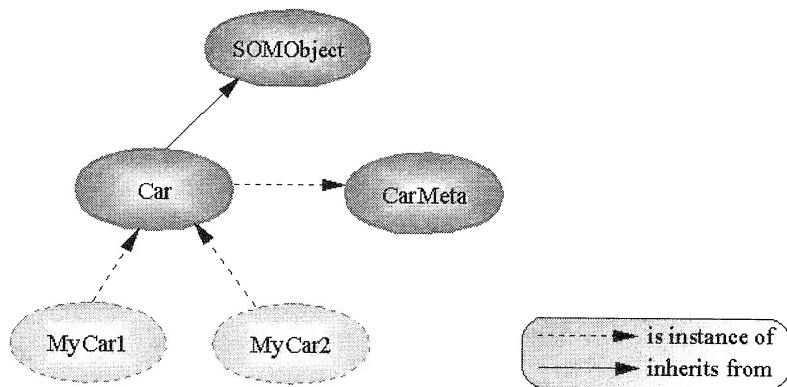
## The SOM Technology

SOM introduces a new way of managing classes. The most important difference between a C++ class and a SOM class is that a SOM class is a **run-time** entity. The run-time representation of a class is called a *class object* as opposed to a simple object, which is an *instance* of the class. You must view a class as an entity that can dynamically be altered. For example, you can add methods to a SOM class “on the fly.”

In SOM, all objects must be instances of some class; so must a class object: the class of a class object is called a *metaclass*. Just as classes define the methods and variables for the instances of that class, a metaclass defines the methods and variables of the class object. Getting confused? Do not panic. Let us take a simple example: You define a class *Car*, and create two instances out of it, *MyCar1* and *MyCar2*. The *Car* metaclass is called *CarMeta*. Figure 158 illustrates the relationship between the classes and class object.

A metaclass can be seen as a “supervisor” class. It usually defines methods for managing class instances, such as creation or deletion. If you do not provide a metaclass for a SOM class, SOM creates a default metaclass which provides basic operations such as retrieving the size of an instance, the list of methods supported by the class, or the name of the class.

The concept of metaclasses brings a lot of power to SOM, and considerably extends the traditional C++ object model. Since SOM allows you to create your own metaclass for a class, you can manage, for example, the creation and deletion of class instances (see “SOM Metaclasses” on page 437).



**Figure 158.** SOM Classes and Metaclasses Relationship

SOM comes with a metaclass framework, which provides utility metaclasses such as:

- ❑ **SOMMSingleInstance:** This metaclass allows you to control the creation of class instances. If a class has this metaclass, then one and only one instance of the class can be created.
- ❑ **SOMMTraced:** This metaclass provides facilities for tracing methods invocations.

See the *SOM Programming Guide* for more information on the Metaclass Framework.

## The SOM Run-Time Environment

The SOM run-time environment is based on three SOM classes:

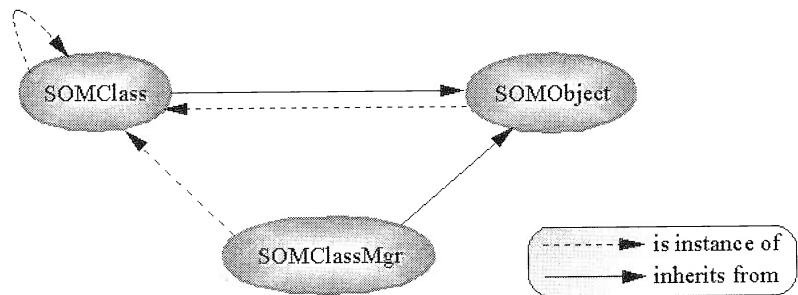
- ❑ **SOMObject:** This class defines the basic behavior of all SOM objects. Every SOM Class must inherit from SOMObject, either directly or indirectly by deriving from a class that has SOMObject as an ancestor class.
- ❑ **SOMClass:** This class defines the basic behavior of all SOM classes. Every metaclass must inherit from SOMClass. SOMClass has the unique characteristic of being its own metaclass. Hence, SOMClass is an instance of itself!

- ❑ **SOMClassMgr**: This class has the responsibility for maintaining a registry of a SOM classes that exist in the current process, and assist in loading and unloading dynamic libraries.

Figure 159 summarizes the relation among those three classes.

At the initialization of the SOM run-time, four objects are created:

- ❑ The SOMClass class object
- ❑ The SOMObject class object
- ❑ The SOMClassMgr class object
- ❑ An instance of SOMClassMgr, called *SOMClassMgrObject* and often referred as to the *SOM Class Manager*.



**Figure 159.** SOMClass, SOM Object, and SOMClassMgr Relationship

## SOM by Example

We propose to show you what you need to do in order to create a SOM version of your calculator class. You first describe the interface of the class using the IDL, then use the SOM compiler to generate the correct bindings for this class, complete the implementation of the class, and finally write some client code using this class.

### Defining the Interface of the Calculator Class

The IDL definition of your calculator class interface can be written as follows:

```
#include <somobj.idl>
interface calculator: SOMObject { 1
    attribute short operand1; 2
    attribute short operand2; 3
    short add (); 4
    #ifdef __SOMIDL__
```



```

implementation {
    releaseorder:  _get_operand1, _set_operand1,
                  _get_operand2, _set_operand2,
                  add; ⑤
};
#endif
};

```

The code is structured as follows:

1. **Interface section:** You need to declare the class interface (①). All SOM classes inherit from *SOMObject*, whose definition is stored in the *somobj.idl* file.
2. **Attribute section:** Your class has two attributes: *operand1* (②) and *operand2* (③). In SOM, an attribute is an instance variable that can be manipulated only through accessors. Declaring an attribute automatically causes SOM to generate specific *get* and *set* methods for this attribute. For example, SOM generates the following code for the *operand1* attribute:
 

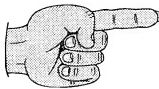
```

short _get_operand1(...);
void _set_operand1(..., in short operand2);

```
3. **Method section:** You define one method, called *add* (④) that adds *operand1* and *operand2*, and returns the result of the operation.
4. **Implementation section:** In this section, you define the release order (⑤) for your class. The release-order statement is specific to SOM, thus you must insert it in a `#ifdef __SOMIDL__ / #endif` statement if you want your IDL to be CORBA-compliant.

The release-order list must include every method defined in the class, including the attribute accessors.

#### Read This



The release-order list is the key to RRBC. Once your class is used by client code, you must not change the order of the items in the list:

- ☐ If a method is removed from the class, then it must not be removed from the list
- ☐ If a new method is added to the class, it must be added at the end of the list

## Generating C++ Bindings with the SOM Compiler

Once you have defined the interface using IDL, you must decide which language you will use for implementing your calculator. In this example, we chose to implement our SOM class in C++. The SOM compiler can generate three files for C++ bindings:

1. **CALCULATOR.XH:** This header file contains the *usage bindings*, that is the definition of the class in a particular language. This file must be included by any C++ client code to access the class methods. This file is included by `calculator.xih`.

The `sc -sxh calculator.idl` command generates this file.

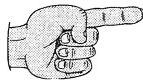
2. **CALCULATOR.XIH:** This header file contains the *implementation bindings*, that is the internal definition of the class in C++. You should not edit or modify this file, which is included by `calculator.cpp`.

The `sc -sxih calculator.idl` command generates this file.

3. **CALCULATOR.CPP:** This source file contains the *implementation template* of the calculator methods. You must modify this file to implement for example the `add()` method.

The `sc -sxc calculator.idl` command generates this file.

#### Read This



You can generate those files at once by using the following command:

```
sc -sxh;xih;xc calculator.idl.
```

**Note:** you can generate those files at once by using the following command: `sc -sxh;xih;xc calculator.idl`.

## Completing the Class Implementation

The SOM compiler generates the following stub for the `add()` method:

```
SOM_Scope short SOMLINK add(calculator *somSelf, Environment *ev)
{
    calculatorData *somThis = calculatorGetData(somSelf);
    calculatorMethodDebug("calculator", "add");
    /* Return statement to be customized: */
    { short retVal; return (retVal); }
}
```

Let us have a closer look at this definition:

1. **SOM\_Scope and SOMLINK:** These keywords are reserved to SOM and are not of interest to the programmer
2. **somSelf** is a pointer to the target object, in our case to a calculator instance. The target object is always the first parameter of any SOM method, although you did not specify it in the IDL specification of the class.

3. **ev** is a pointer to an *Environment* structure. You **must** specify this parameter when using a SOM method. The environment structure is used by SOM to return exception information from the method to the caller if a problem occurs.
4. **somThis** is a pointer that gives you direct access to the value of the attributes you defined in the class, respectively ***operand1*** and ***operand2***. You can use it as follows: *somThis->operand1* or *\_operand1* (if the `VARIABLE_MACRO` macro is set).

Modify the `add ()` method as follows:

```
SOM_Scope short SOMLINK add(calculator *somSelf, Environment *ev)
{
    calculatorData *somThis = calculatorGetData(somSelf);
    calculatorMethodDebug("calculator", "add");

    short retVal;
    retVal = somThis->operand1 + somThis->operand2 ;
    return retVal;
}
```

## Writing a Client Application

Your SOM class is now ready to be used. You need to write a “main” program that creates an instance of the calculator class, initializes the ***operand1*** and ***operand2*** attributes, adds them, and prints out the result. Using the bindings provided by SOM, the client program can be written either in C or C++. If you want to implement a C client program, then you must generate the C usage bindings by invoking the SOM compiler as follows: `sc -sh calculator.idl`.

You can use your SOM class from any tool providing a SOM wrapper, such as VisualAge for Smalltalk, V3.0, which allows you to read a class interface description written in the IDL, create a part out of it, and use as any other class.

If you use the C++ bindings, then the `main.cpp` client program can be written as follows:

```
#include "calculator.xh" ❶
#include <iostream.h>

int main (int argc, char *argv[])
{
    Calculator *myCalc = new Calculator; ❷
    Environment *ev = somGetGlobalEnvironment (); ❸
    short result;

    myCalc->_set_operand1 (ev, 5); ❹
```

```

myCalc->_set_operand2 (ev, 3);

result = myCalc->add(ev);
cout << " Result of the addition is: " << result << endl;
myCalc->somDumpSelf(0); 5

delete myCalc; 6
}

```

- 1 The client program includes the usage bindings file, here calculator.xh.
- 2 An instance of calculator is created. Notice that the new operator is used.
- 3 The environment pointer variable is initialized.
- 4 The *operand1* and *operand2* attributes are initialized. Notice that you can manipulate the attributes value only through their accessors.
- 5 The somDumpSelf method prints out information about the myCalc instance. You might override this function to suit your own needs.
- 6 The myCalc instance is destroyed. You can use myCalc->somFree() to delete an instance. Both instructions are equivalent.

To compile and link the following program:

1. Open an MS-DOS window
2. Go to the directory where you have created the calculator sample files.
3. Type: `icc -I. /Gm /Fecalctest calculator.cpp main.cpp somtk.lib`

Hopefully, running the calctest executable produces the following output:

```

> calctest
{An instance of class calculator at address 0066E3F4}
Result of the addition is: 8
>

```

Congratulations! You have written your first SOM program! Now, we are going to show you how to create a DLL containing your calculator class.



## Creating a SOM DLL

In this section, you use the WorkFrame to create the calculator DLL. The following explanations assume that your code is organized as follows:

- ❑ The IDL source, usage bindings, implementation bindings, and implementation files are located in D:\VR\Sources\Misc\SOM\dll.
- ❑ Client program files are located in D:\VR\Sources\Misc\SOM.

We also assume that you are familiar with the WorkFrame component.

### ***Specifying the DLL Initialization Function***

When you create a SOM DLL, you must specify an initialization function. This function is required by the SOM Class Manager to create and register the class objects for all the classes contained in the library when it loads the class library. For each class in the class library, you must invoke the `<ClassName>NewClass` function.

For the calculator DLL, you must create a `sominit.cpp` file with the following contents in the D:\VR\Sources\Misc\SOM\DLL directory:

```
#include "calculator.xh"
void _System SOMInitModule( long majorVersion, long minorVersion,
                           string className)
{
    CalculatorNewClass( Calculator_MajorVersion,
                       Calculator_MinorVersion);
}
```

The `CalculatorNewClass` function is created by SOM when you generate the `calculator.xh` file (usage bindings). Note that the parameters supplied to the initialization are not needed in most cases.

### ***Creating a Module Definition File***

You need to create a module definition file (or “def” file) that contains all exported symbols for the library. For each class in the library, you must export the three following functions:

- ❑ `<ClassName>ClassData`
- ❑ `<ClassName>CClassData`
- ❑ `<ClassName>NewClass`

You can create the “def” file in three ways:

1. Write it yourself!

2. Use the SOM def emitter to create the def file from the IDL by issuing the following command: `sc -sdef calculator.idl`, which produces the following `calculator.def` file:

```
LIBRARY calculator
EXPORTS
_calculatorCClassData
_calculatorClassData
_calculatorNewClass@8
```

3. Use the Make Def action in WorkFrame to create the def file for you automatically.

The Make Def action calls the CPPFILT utility that parses your object files and creates a def file from the parsing results. In this sample, we use this solution, which prevents the user from modifying the def file manually after it was generated by the SOM compiler (for example, to add the SOMInitModule function to the list of exported functions). Read “Customizing the Calc DLL project” on page 434 for details on WorkFrame setup.

## Creating the Workframe Projects

You need to create two projects, one to manage the DLL-related files (Calc DLL), the other to manage the client program (Calc Test).

To create the Calc DLL project:

1. Double-click on the **WorkFrame IDE** icon, located in the IBM VisualAge for C++ for Windows folder
2. Select **Create new project**
3. Select **Default Project** application from the available project list
4. Click on **Next>>**
5. In the dialog window, enter the following data:

<b>Project title</b>	Calc DLL
<b>Source file directory</b>	D:\VR\Sources\Misc\SOM\d11
<b>Where to create the project files</b>	D:\VR\Sources\Misc\SOM
<b>Project file name</b>	CalcDLL

6. Click on **Done** to create the project

Repeat these steps to create the Calc Test project with the following data:

<b>Project title</b>	Calc Test
<b>Source file directory</b>	D:\VR\Sources\Misc\SOM

<b>Where to create the project files</b>	D:\VR\Sources\Misc
<b>Project file name</b>	CalcTest

You are now ready to set up the compile and link settings for those two projects.

### *Customizing the Calc DLL project*

First, you need to change the project settings. To do so,

1. Open the Calc DLL project.
2. Choose **View**→**Settings**→**Target** from the project toolbar, and enter the following information:

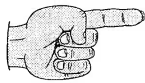
**Target Name** somcalc.dll

**Makefile** Makefile

3. Switch to the Environment notebook page and add the following environment variable:

INCLUDE=.;%INCLUDE%

#### **Read This**



You do not have to setup the INCLUDE variable if the current directory (.) is already declared in the INCLUDE system environment variable.

4. Click on **OK** to commit the changes.

Then, you need to set up the different WorkFrame actions:

1. Choose **Options**→**Build normal** from the project toolbar.
2. From the list of available actions, select the following:

- Compile
- Link
- Make Def File

MakeMake uses the CPPFILT utility to create a module definition file (“def” file) from the object files included in the library.

- Make Exp and Lib File.

The IBM Librarian Manager (ilib) uses the .def file to create an export file and a library.

3. Switch to the Project notebook page and make sure that Use Build Settings from Parent Project is NOT set.
4. Click on **OK** to commit the changes.
5. Repeat Steps 2 to 4 for the **Rebuild all** action.

6. Choose **Options**→**Compile** from the project toolbar, and set the following:
  - On the Processing page: you must perform **compile** only, and the result of compilation is a DLL.
  - Click on **OK** to commit the changes.
7. Choose **Options**→**Link**, and set the following:
  - On the File Names page, enter **somtk.lib** in the Add Libraries entry field.
  - On the Definition:Run-file subpage, select Dynamic Link Library.
  - Click on **OK** to commit the changes.

### ***Customizing the CalcTest project***

First, you need to change the project settings. To do so,

1. Open the Calc Test project.
2. Choose **View**→**Settings**→**Target** from the project toolbar, and enter the following information:
 

**Project Target** calctest.exe

**Makefile** Makefile
3. Switch to the Environment notebook page and add the following environment variables to the project:

Variable	Variable Use
LIB=.\\d11;%LIB%	This variable is used at link time to find the somcalc.lib file.
INCLUDE=.\\d11;%INCLUDE%	This variable is used at compile time to find the calculator.xh file.
PATH=.\\d11;%PATH%	This variable is used at run time to find the somcalc.dll file.

4. Click on **OK** to commit the changes.

Now, you must customize the WorkFrame actions:

1. Choose **Options**→**Build normal** from the project toolbar, and select the following actions:
  - Compile
  - Link
2. Switch to the Project page, and make sure the Pass Build Settings to Subprojects option is NOT selected.



3. Click on **OK** to commit the changes.
4. Repeat Steps 2 to 3 for the **Rebuild all** action.
5. Choose **Options**→**Compile**, and set the following:
  - On the Processing notebook page, set the processing step to compile only and the target of compilation to EXE.
  - On the Object notebook page: set the library linkage to **dynamic**. Since we use the IBM Open Class library in the client program (iostream), we must set the library selection to multithread (equivalent to /Gm).
  - Click on **OK** to commit the changes.
6. Choose **Options**→**Link** and set the following:
  - On the File Names notebook page, add **somtk.lib somcalc.lib** to the Add Libraries field.
  - On the Definition:Run-file notebook subpage, select **EXECUTABLE** as the run-file.
  - Click on **OK** to commit the changes.

### ***Building the Calculator Test***

To create the Calculator DLL as well as the client program, just select **Project**→**Build normal** from the CalcTest project toolbar.

## **Modifying the SOM DLL**

You now modify the Calculator DLL and check that RRBC is a reality. As explained in “Why is recompilation needed for C++ Libraries?” on page 423, if you add an instance variable at the beginning of a C++ class definition, then binary compatibility is broken.

Let us modify our sample calculator.idl by adding a result attribute to store the result of the calc operation. The calculator.idl now looks like this:

```
#include <somobj.idl>
interface calculator: SOMObject {
    attribute short result; 1
    attribute short operand1;
    attribute short operand2;
    short add ();
    #ifdef __SOMIDL__
    implementation {
        releaseorder:  _get_operand1, _set_operand1,
                       _get_operand2, _set_operand2,
                       add,
                       _get_result, _set_result; 2
    }
    #endif
}
```

```
};
#ifdef ...
};
```

The attribute ❶ can be added anywhere in the code, but the corresponding accessors must always be added at the end of the release order list ❷. This is the golden rule of RRBC. To create a new DLL, we must regenerate the various bindings, `.xh`, `.xih`, and `.cpp`. What is going to happen to the `add()` method that we have customized? Nothing. The SOM compiler is smart enough to merge to existing code with the newly created one.

To test RRBC, open the Calc DLL project and choose **Project→Make**. Then open the CalcTest project and run the `calctest.exe` program. The output should still be the same, and  $5 + 3$  still equals 8.

## Advanced Features

In this section, we give you some deeper information on the SOM technology. You may want to skip this section if you think you know enough about SOM and go directly to “Understanding Direct-to-SOM” on page 443.

### SOM Metaclasses

The concept of metaclasses, and particularly the ability to customize metaclasses definition, brings a lot of power to SOM. Using metaclasses, you can solve problems that are difficult to cope with in traditional C++. As an example, we later explain how to create a metaclass that counts the number of instances of a given class, which can be a tricky task in C++.

In SOM, you cannot create a constructor that takes parameters. Hopefully, you can turn the problem around by providing a metaclass that creates the instances for you, and then perform any initialization task, such as setting a default value for the instance variables.

Let us now create a metaclass for the calculator class: this metaclass is derived from the `SOMMTraced` metaclass (which is part of the SOM Metaclass Framework) and has the following interface:

- ❑ An ***instanceCount*** attribute, that contains the number of existing instances of class calculator.
- ❑ A ***createCalc*** method, that creates an instance of calculator and increments `instanceCount` by 1. This function takes two parameters, `op1` and `op2`, that are used to initialize the value of the calculator operands.

- A `destroyCalc` method, which deletes a given instance.

The interface of the `metaCalc` metaclass is described in IDL as follows:

```
interface metaCalc : SOMMTraced {
    attribute short instanceCount;
    calculator createCalc (in short op1, in short op2);
    void destroyCalc (in calculator calcToDestroy);

#ifdef __SOMIDL__
    implementation {
        releaseorder: _set_instanceCount, _get_instanceCount,
                      createCalc,
                      destroyCalc;
    };
#endif
};
```

#### Read This



You can create the definition in the same .idl file as the one that already contains the definition for the calculator class.

You must now declare `metaCalc` as the metaclass for the calculator. You do this by inserting a **`metaclass=`** modifier in the implementation section of the calculator IDL definition:

```
#ifdef __SOMIDL__
    implementation {
        releaseorder:....
        metaclass = metaCalc;
    };
#endif
```

The next step is to regenerate all the bindings using the SOM compiler, and modify the stub functions for the `createCalc()` and `destroyCalc()` methods as follows:

```
SOM_Scope calculator* SOMLINK createCalc(calculatorMeta *somSelf,
                                         Environment *ev, short op1,
                                         short op2)
{
    calculatorMetaData *somThis = calculatorMetaGetData(somSelf);
    calculatorMetaMethodDebug("calculatorMeta","createCalc");

    calculator* aCalc = somSelf->somNew(); 1
    aCalc->_set_operand1 (ev, op1); 2
    aCalc->_set_operand2 (ev, op2);
```

```

        somThis->instanceCount += 1; ❸
        return (aCalc);
    }

SOM_Scope void  SOMLINK destroyCalc ( calculatorMeta *somSelf,
                                     Environment *ev,
                                     calculator* calcToDestroy)
{
    calculatorMetaData *somThis = calculatorMetaGetData(somSelf);
    calculatorMetaMethodDebug("calculatorMeta","destroyCalc");

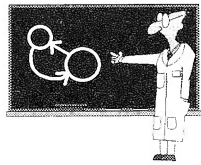
    calcToDestroy->somFree(); ❹
    somThis->instanceCount -=1; ❺
}

```

Let us have a closer look at the code for the createCalc() method:

1. First, an instance of the calculator class must be created (❶). This is accomplished by using the somNew() method. When using somNew, you create an instance of the given class, initialize the object that has been created, and then return a pointer to that object.

#### Technical Information



When you use somNew, you allocate some memory for the instance. If you already have some memory allocated, and you want to use this space to store an object, you can use the somRenew method. It is then your responsibility to ensure that the block of storage given as a parameter to somRenew is large enough to hold the new instance.

2. Then, the calculator attributes, **operand1** and **operand2** are initialized (❷).
3. Finally, the number of instances is incremented by 1. (❸)

The deleteCalc() method is rather simple: the instance passed in parameter is deleted by using the somFree (❹) method (which works together with somNew) and the number of instances is decremented by 1 (❺).

Now, you can rebuild the somcalc.dll by choosing **Project→Make** from the Calc DLL project.

### Modifying the Original Client Code

The original client code needs to be modified so that the metaclass is used to create instances of the calculator part, using the createNew function that you implemented. This code is presented below:

```
#include <calc.xh>
#include <iostream.h>
void main (int argc, char *argv[])
{
    calculatorMeta *MetaCls; ❶
    calculator *calc1, *calc2;
    Environment *ev;
    int operand1, operand2, result;

    ev=somGetGlobalEnvironment();
    MetaCls = calculatorNewClass (0, 0); ❷
    MetaCls->_set_instanceCount (ev, 0);

    calc1 = MetaCls->createCalc (ev, 8, 4); ❸
    calc2 = MetaCls->createCalc (ev, 5, 5);

    cout<<"Instances: " << MetaCls->_get_instanceCount(ev) << endl; ❹
    cout<<"Instance Size: " << MetaCls->somGetInstanceSize() << endl;
    cout<<"Class Name: " << MetaCls->somGetName() << endl;
    cout<<"Addition Result Class1: " << calc1->add(ev) << endl;
    cout<<"Addition Result Class2: " << calc2->add(ev) << endl;

    MetaCls->destroyCalc (ev, calc1); ❺
    MetaCls->destroyCalc (ev, calc2);
    MetaCls->somFree();
    return;
}
```

1. A metaCls object is declared (❶) and then instantiated (❷) using the calculatorNewClass() method. The <className>NewClass method creates the class object (unless it already exists), creates class objects for the ancestors and the metaclass of the class, and returns a pointer to the newly created class object (a class object is an instance of the metaclass, which explains why we assign the result of the calculatorNewClass call to the metaclass metaCls object). The number of instances is initialized to 0.
2. Then, we create two instances of the calculator class, using the createNew method (❸). The parameters supplied to the function are used to initialize both calculator operands.
3. Then, we use the metaclass features to request some information from the class, such as the size of an instance or the name of the class. The number of instances, maintained by the metaclass itself, is also requested. We also print out the result of the addition for each addition.

4. Finally, all objects are deleted from memory. The metaclass is used to delete each calculator instance. The metaclass itself is explicitly deleted by using the `somFree` method.

When you execute this program, you get the following output:

```
Instances: 2
Instance Size: 10
Class Name: calculator
Addition Result Class1: 12
Addition Result Class2: 10
```

## Functions Overriding

One of the important aspects of OOP is the ability of a subclass to replace the implementation of an inherited method with a new implementation that better suits its needs. Such ability is referred to as *method overriding*. SOM provides you with two methods that are likely to be overridden in each SOM class that you create:

- ❑ `somPrintSelf`: This method is intended to provide a brief description of an object.
- ❑ `somDefaultInit`: This method is intended to provide a default initializer for the instance variables of a class.

Previously, you had to initialize the ***instanceCount*** attribute to 0 in the client code before using the `createNew` method. Now, you slightly modify the metaclass definition to use the `somDefaultInit` function to perform this task.

You begin by modifying the IDL definition of the `calculatorMeta` metaclass to specify that the `somDefaultInit` method is overridden. This is done by using the `override` keyword that must be inserted in the implementation section of the interface definition. The `init` keyword is used to indicate to the `xc` emitter, which produces the `.cpp` binding file for the calculator class, that a specific stub must be generated for the `somDefaultInit` method.

```
#ifdef __SOMIDL__
    implementation {
        releaseorder:...
        somDefaultInit:override, init;
    }
#endif
```

Next, you must regenerate the bindings for the calculator class using the `sc -sxh;xih;xc calculator.idl` command. The SOM compiler produces the following stub for the `somDefaultInit` method:

```
SOM_Scope void SOMLINK somDefaultInit( calculatorMeta *somSelf,
                                       somInitCtrl* ctrl)
{
    calculatorMetaData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    calculatorMetaMethodDebug("calculatorMeta","somDefaultInit");
    calculatorMeta_BeginInitializer_somDefaultInit;
    calculatorMeta_Init_SOMMTraced_somDefaultInit(somSelf, ctrl);
    /*
     * local calculatorMeta initialization code added by
    programmer
     */
    → Your code goes here...
}
```

The next step is to customize this method for your needs by providing your initialization code. What you need is to set the initial ***instance-Count*** attribute to 0 by adding this line:

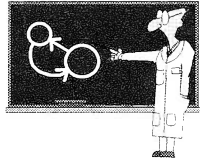
```
somThis->instanceCount = 0;
```

Then, you must update the client program to remove the unnecessary statement that initializes the instanceCount attribute to zero, and rebuild the CalcTest project. The program output should still be the same!

## Attributes vs. Instance Variables

When you define a SOM attribute, such as attribute short index, get and set accessors are automatically created for you by the SOM compiler. As an alternative to attributes, you can use instance variables. Instance variables are defined in the implementation section of an IDL definition. It is then your responsibility to provide the necessary accessors.

### Technical Information



The default set method generated by the SOM compiler does a “shallow copy” of objects. In other words, if an object containing a pointer to another object is copied, then only the pointer is copied (as opposed to “deep copy,” where the pointed object is also copied). This might not be suitable in some situations, especially when you deal with strings and pointers, or in a distributed environment. In such cases, you should use the `noset` modifier to specify that the set method should not be defined by default, but rather that you want to implement it yourself. The SOM compiler then generates a stub for the set method that you can customize to your needs.

As a rule of thumb, any instance data that must be accessible to client programs or from a subclass should be defined as an attribute. Any instance data that is relevant only to the class itself should be defined as an instance variable, along with appropriate methods to access it.

## Understanding Direct-to-SOM

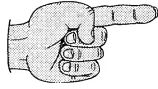
The conventional way of using SOM classes implies that you write the class definition in IDL, use the SOM compiler to create the necessary bindings, and finally implement the class in the language of your choice. If you already have C++ classes that you would like to reuse as SOM classes, you must rewrite the classes interface in IDL, and rewrite the class implementation (inspired from the existing code).

With the DTS feature of the VisualAge for C++ compiler, you can take advantage of SOM while using most of the features of the C++ language such as templates, operators, constructors with parameters, or static members. A C++ class is converted to a DTS class by inheriting from the `SOMObject` class, defined in the `som.hh` header file. From the DTS class definition, you can then generate the equivalent IDL definition by compiling the header file with the `/Fs` compiler switch.

A major inhibitor to RRBC with C++ is the fact that so much information about an object is statically compiled into the client code, and in particular, the location of instance data and virtual table pointers. When you are working in DTS mode (as opposed to native mode for straight C++), data layout and method calls for the DTS class are performed using the SOM API instead of the native C++ API.



When you run a program using a SOM object, the SOM run time is used to create and manipulate the class object and its instances.

**Read This**

All DTS classes definition *must* be stored in a header file with extension `.hh`. IDL generation does not work if you do not obey this rule.

## Restrictions Imposed by DTS

When you use DTS classes, you must keep in mind the following rules:

1. A DTS class can inherit only one copy of a given ancestor class. This condition is met if:
  - Each DTS ancestor has a single inheritance path.
  - In case of a multiple inheritance path, the base classes are inherited as virtual, for example  
`class myclass: public virtual baseClass`

The VisualAge for C++ compiler issues a fatal error if this condition is violated.

2. You cannot mix DTS classes and native C++ classes in a single compilation unit. As a rule of thumb, you should never include header files with extensions `.xh` and `.hh` in a same C++ source file.

## DTS by Example

In Chapter 8, “Creating Nonvisual Parts”, we created the `AMortgage-Calculator` class interface part, which is used to calculate a maximum mortgage value, according to buyer income, interest rate, and mortgage duration. We now want to transform this C++ class into a DTS class.

This can be achieved in two ways:

1. Inherit from `SOMObject`

In this case, you should modify the declaration of the `AMortgage-Calculator` class (in a `vrssimc.hh` file) as:

```
#include <som.hh>
class AMortgageCalculator: public SOMObject {
    ...
};
```

2. Use the `SOMAsDefault` compiler pragma

The `SOMsDefault` pragma is used to tell the compiler if it should treat classes as C++ or DTS classes. Use `#pragma SOMsDefault(on)` to turn DTS mode on:

```
#pragma SOMsDefault (on)
class AMortgageCalculator
{
    ...
}
#pragma SOMsDefault (off)
```

You do not need to include the `som.hh` file explicitly, it is done automatically the first time the `SOMsDefault` pragma is on. You should use the `SOMsDefault` pragma when you want to keep a single source and use a macro to switch the DTS mode on or off. You can then easily choose to generate a C++ or DTS C++ library. For example, you can change the previous class definition as:

```
#ifdef DTS_MODE_ON
    #pragma SOMsDefault (on)
#endif
class AMortgageCalculator
{
    ...
};
#ifdef DTS_MODE_ON
    #pragma SOMsDefault (off)
#endif
```

Then, you need only use the `/DDTS_MODE_ON` compiler flag each time you want to produce a SOM object rather than a C++ object out of the `AMortgageCalculator` class.

### Warning



You may also use the `/Ga` compiler flag to turn the DTS mode on. Be cautious when using this flag: it does not work in most cases. The rationale for this is simple: when you turn `/Ga` on, you tell the compiler to “SOMify” any classes it finds in the compilation unit: This of course includes the classes you have defined, but also any class defined in the header files you are using. In other words, if your source file includes `istring.hpp` then the compiler builds a SOM-enabled version of the `IString` class *interface*. Unfortunately, VisualAge for C++ does not come with a SOM-enabled version of the IBM Open Class Library. Thus, the linker fails trying to find a SOM version of the `istring` class *implementation*. We therefore highly recommend that you do not use the `/Ga` compiler flag.

Up to this point, using DTS seems like a pretty simple task: only a few changes to the original code are required to inherit from the features SOM brings to OOP, and in particular binary compatibility. Actually, the level of change required in the original C++ code depends heavily on who are the targeted users of those classes:

1. If the targeted users are using only DTS C++, then you provide them the DTS header files (.hh) and a dynamic link library. In such a case, only minimal changes are required to the C++ class definition.
2. If the targeted users wish to use your library in a CORBA environment, then they need the IDL definition. The C++ compiler has a compiler switch (/Fs+) that allows generating the IDL equivalent to the C++ interface. Unfortunately, this IDL is difficult to interpret, especially because the compiler mangles all names. For example, if we generate the IDL definition for the AMortgageCalculator class, the interface name is changed to `zazmortgagecalculator`—not a “user friendly” name.

In both cases, you must use pragma compiler directives to control the output of the compiler.

## Controlling the Compiler Output with Pragmas

The C++ compiler defines a set of pragmas that you can use to control name mangling, but also to define a SOM attribute, or a SOM meta-class. SOM pragmas are described in detail in the *VisualAge for C++ Programming Guide* (the IBM System Model chapter). We do not intend to reexplain all of them here, but rather to show how we used them when converting the AMortgageCalculator class to DTS.

Here is the header file for the AMortgageCalculator class once the necessary pragmas have been added:

```
class AMortgageCalculator
{
    #pragma SOMClassName (AMortgageCalculator, "AMortgageCalculator")
    #pragma SOMNoMangling (on)
public:
    //-----
    // Constructors/Destructors
    //-----
    AMortgageCalculator();
    virtual ~AMortgageCalculator () {};
    //-----
    // Public Member Functions
    //-----
    double estimate();
    //-----
}
```

```

// Data Members
//-----
double income;
#pragma SOMAttribute(income)

double rate;
#pragma SOMAttribute (rate)

double years;
#pragma SOMAttribute (years)

double estimation;
#pragma SOMAttribute (estimation)

#pragma SOMReleaseOrder( estimation,
                           calculate,
                           income,
                           rate,
                           years);
};
#pragma SOMAsDefault (off)
#pragma SOMNoMangling (off)

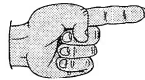
```

## Controlling Name Mangling

Several of the pragmas we used are needed to control name mangling:

- ❑ **SOMClassName:** Use this pragma to specify the SOM name of your class.

### Read This



In the scope of the class, you can replace the name of the class by a star: `#pragma SOMClassName(*, "AMortgageCalculator")`.

- ❑ **SOMNoMangling:** Use this general pragma to tell the compiler that it should not mangle C++ names of methods, static member functions, and instance data.
- ❑ **SOMDataName:** Use this pragma to specify the SOM Name of a class data member. Because SOM is case insensitive, names of any methods or data members should be distinguishable, regardless of case.

**Warning**

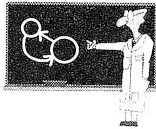
Normally, you would not use the `SOMDataName` and the `SOMNoMangling` pragmas together, but we found a case where you have to: if the attribute has capitalized letters, the SOM compiler mangles its name anyway, even if the `SOMNoMangling` pragma is in effect. This situation happens for the ***simulationResult*** attribute. Therefore, anytime you have an attribute with capitalized letters, you must use the `SOMDataName` pragma. The `SOMNoMangling` pragma works fine with capitalized method names.

**Creating a SOM Attribute**

The `SOMAttribute` pragma is used to define a SOM attribute. The SOM compiler generates the necessary get and set methods for accessing. This pragma accepts several keywords, among which are:

- ❑ **noget:** The C++ compiler does not generate a body for the attribute's get method. You must provide a body for the get method, and you must name it `_get_<attributeName>`.
- ❑ **noset:** This is equivalent to `noget`, but for the set method. The method must be named `_set_<attributeName>`.
- ❑ **readonly:** The C++ compiler does not generate any set method. If you define a `_set_<AttributeName>`, the compiler throws an error.

By default, attributes are generated as private, and can be manipulated only through their accessors.

**Technical Information**

If you do not declare an attribute with the `SOMAttribute` pragma, this attribute is considered by the C++ compiler as an instance variable. See "Attributes vs. Instance Variables" on page 442 for more information on this topic.

**Handling Constructors and Destructors**

The `AMortgageCalculator` defines a default constructor, as well as a virtual destructor. The C++ compiler automatically maps the default constructor to the `somDefaultInit` method, and the destructor to the `somDestruct` method. They are defined as follows in the generated IDL:

```
somDefaultInit:
public, override, init, cxxdecl="AMortgageCalculator()";
somDestruct: public, override, cxxdecl="virtual
~AMortgageCalculator()";
```

## Specifying the Class Release Order

The `SOMReleaseOrder` pragma allows you to specify the release order for your class. You may omit to specify the release order for a class, as DTS builds a default. However, this option forces you to be extremely cautious when adding methods or data members to your DTS class: You must ensure they are always added at the *end* of the public, protected, or private sections of the class definition. Thus, we strongly recommend using the `SOMReleaseOrder` pragma.

You may enter the release order yourself or use the `/Fr` compiler option to generate the release order to the standard output. You then just cut and paste the compilation result into the DTS header file. You must pass the name of the class as a parameter to `/Fr`. For example, you should invoke `icc /Fr"AMortgageCalculator" vrssimc.hh` to get this result:

```
/* AMortgageCalculator */
#pragma SOMReleaseOrder( \
/* 1 */_get_estimation() const,\
/* 2 */_set_estimation(double),\
/* 3 */_estimate(),\
/* 4 */_get_income() const,\
/* 5 */_set_income(double),\
/* 6 */_get_rate() const,\
/* 7 */_set_rate(double),\
/* 8 */_get_years() const,\
/* 9 */_set_years(double))
```

Note that you can either specify the attribute name directly in the release order, or in its accessors.

Once you have defined all necessary pragmas, you can generate the class interface in IDL by invoking: `icc /Fs+ vrpsimul.hh`. The interface of the class defined in the resulting IDL is the same as in the DTS header file, and you can use it easily from another language.

## Generating DTS Headers from IDL

The SOM compiler also comes with a DTS emitter, which creates DTS header files, instead of the traditional C++ bindings generated by the `xc` emitter. This approach might be interesting if you do not have existing C++ code, but you want to use SOM in a more transparent way than by using the C++ bindings.

As a example, DTS takes care of the SOM environment variable for you. As a result, you can transparently use a DTS class as if you were dealing with a pure C++ class.

Let us write the `AMortgageCalculator` class interface in IDL:

```

#include <somobj.idl>
interface AMortgageCalculator: SOMObject
{
    attribute double income;
    attribute double rate;
    attribute double years;
    attribute double estimation;
    double estimate ();

#ifdef __SOMIDL__
    implementation {
        dtsclass; ❶
        releaseorder: _get_estimation, _set_estimation,
                      estimate,
                      _get_income, _set_income,
                      _get_rate, _set_rate,
                      _get_years, _set_years;

        somDefaultInit: override, init, cxxdcl="AMortgageCalculator;";
        // The constructor is mapped to somDefaultInit ❷
        somDestruct: override, cxxdcl="virtual ~AMortgageCalculator;";
        // The destructor is mapped to somDestruct
    };
#endif
};

```

This IDL uses special modifiers to indicate the SOM compiler that the generated class is a DTS class:

- ❶ **dtsclass:** The DTS class modifier tells the hh emitter, which generates DTS C++ header files, that the semantics of this class should be interpreted according to C++ rules.
- ❷ **somDefaultInit** and **somDestruct:** These two methods can be overridden with the default constructor and destructor. Note that the `cxxdcl=` is not a standard option. This statement is inspired from the code generated by the VisualAge for C++ compiler when you compile a .hh file to generate IDL (see “Handling Constructors and Destructors” on page 448).

You can use the SOM compiler against this IDL file and use the hh emitter to generate the corresponding DTS header file. To generate DTS headers, issue the following command:

```
sc -shh -mnoqualifytypes vrssimc.idl.
```

The following `vrssimc.hh` file is produced:

```

#ifndef _DTS_HH_INCLUDED_vrssimc
#define _DTS_HH_INCLUDED_vrssimc

/* Start Interface AMortgageCalculator */

```

```

// This file was generated by the IBM "DirectToSOM" emitter for C++
(V1.116)
// Generated at 08/23/96 18:09:47
// The efw file is version 1.57

#ifndef som3AssignCtrl
#define som3AssignCtrl somAssignCtrl
#endif

#include <som.hh>
    #pragma SOMAsDefault(on)
class SOMClass;
    #pragma SOMAsDefault(pop)
    #pragma SOMAsDefault(on)
class SOMObject;
    #pragma SOMAsDefault(pop)
#include <somobj.hh>

class
#if !(defined(SOM_AMortgageCalculator_Class_Source) ||
defined(SOM_DONT_IMPORT_CLASS_AMortgageCalculator))
WIN32_DLLIMPORT
#else
WIN32_DLLEXPORT
#endif
AMortgageCalculator : public ::SOMObject {
    #pragma SOMClassName(*, "AMortgageCalculator")
    #pragma SOMNoMangling(*)
    #pragma SOMCallstyle (idl)
    #pragma SOMAsDefault(off)
public :
    #pragma SOMAsDefault(pop)
    virtual double estimate();

    // The constructor is mapped to somDefaultInit
    AMortgageCalculator();

    // The destructor is mapped to somDestruct
    virtual ~AMortgageCalculator();

    double income;
    #pragma SOMAttribute(income, virtualaccessors)

    double rate;
    #pragma SOMAttribute(rate, virtualaccessors)

    double years;
    #pragma SOMAttribute(years, virtualaccessors)

```



```
double estimation;
#pragma SOMAttribute(estimation, virtualaccessors)
#pragma SOMReleaseOrder ( \
    "_get_estimation", \
    "_set_estimation", \
    "estimate", \
    "_get_income", \
    "_set_income", \
    "_get_rate", \
    "_set_rate", \
    "_get_years", \
    "_set_years")
};
/* End AMortgageCalculator */
#endif /* _DTS_HH_INCLUDED_vrssimc */
```

### ***Using a DTS class from Visual Builder***

A DTS class from Visual Builder can be used the same way you would use a straight C++ class. You must create a public interface for the class, either by writing a VBE file or by using the class browser. Refer to Chapter 10, "More about Visual Builder...", on page 353 for more information.

The following restriction apply: Only class interface parts can be used from Visual Builder in DTS mode. Remember that a DTS class can only inherit from another DTS class. Similarly, a nonvisual part must inherit from `IStandardNotifier`, either directly or indirectly. Unfortunately, no DTS version of the `IStandardNotifier` class is provided with the VisualAge for C++ product. Therefore, you cannot build a DTS class out of a nonvisual part.

# 13

## More about CDF...

*The transition to component-based software will change the way we buy and build systems and what it means to be a software engineer.*

-Dave Thomas,  
President Object Technology International (March, 1995)

The Compound Document Framework (CDF) allows you to create applications enabled for Object Linking and Embedding (OLE) on the Windows NT and Windows 95 operating systems. This framework is more than just a set of classes, it is also a set of predefined collaborations among those classes that provide you with the major functionality required to create compound document applications.

With the CDF, you can create both OLE containers and OLE servers. In this chapter we introduce you to the framework and describe how to build OLE servers. Furthermore, we show you how to take advantage of the Visual Builder to make your parts OLE-enabled.

We do not intend to cover the compound document technology, and especially OLE in this chapter. Many good books have been written on this subject. Instead, we would like to compare the way you develop a

server component using only the CDF (and programming your OLE application from the ground up) with the way you do it using the Visual Builder.

After reading this chapter you should be able to:

- ☐ Understand the compound document technology as implemented in the framework.
- ☐ Build an OLE server using the CDF.
- ☐ Take advantage of the Visual Builder to build the view and the model of your OLE server.

In this chapter, we do not cover the OLE container programming. For more information on how to build an OLE container, you should consult the VisualAge for C++ Open Class Library User's Guide and the VisualAge for C++ Open Class Library Reference Guide.

## Introducing the Compound Document Technology

For many years, users have been able to create, work with, and save data. Most of the time data saved was simple. For example, when using word processing software, end-users could enter text, edit it, and save it to a file. The resulting files were commonly referred to as *simple documents* because they were composed primarily of text.

Over time, end-users have been looking for more complexity in their document in order to mix graphics, text, sound, and video. To store such complex data in a file, users were using conversion programs to convert their pieces of data in the file format used by the application.

As the number of applications increased, however, each with its own format, it became impossible to write conversion programs for all possible combinations. The idea was to forget about converting data files from one format to another and instead make a space for an embedded application in a container application and then save the contents of both files, each in its own native format in a hierarchically organized file. In this way, many data types could then be saved in the same document file without having to convert between formats. A document that contains other documents or components is known as a *compound document*.

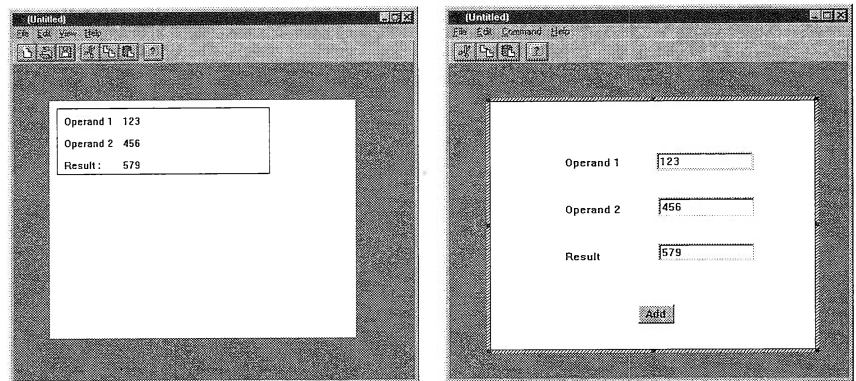
Compound documents are places where components live. They can accept any kinds of data at run time since the data content is managed by the component that owns the data. In contrast, traditional application deal with data types that are known by the application at compile time. In that case, supporting another data type means modifying and recompiling the application code.

Within the last couple of years, compound documents have become a major framework for deploying components on the desktop and across the enterprise. The “document” metaphor provides an intuitive way to organize collections of related components, display them seamlessly in a window, and store, share, and distribute them across the network to other desktops or servers. In this context, the desktop of your operating system is about to become itself a giant compound document that integrates applications, and operating system services and manufacturers are shipping their compound document technology with their operating system.

## Object Linking and Embedding

Among the protocols that have emerged over the years, Object Linking and Embedding (OLE) introduced by Microsoft in 1990 as an add-on for its Windows 16-bit environment, is today one of the most popular.

A few years later, with the release of OLE 2, this compound document protocol has become tightly integrated with the 32-bit Windows NT and Windows 95 operating systems and provides the applications with powerful integration features, such as in-place activation where embedded applications merge their interfaces with the interface of their container when they are activated. Figure 160 shows a calculator component embedded in a container. Notice in the second part of the figure that, when activated, the windows toolbar has changed with the calculator toolbar.



**Figure 160.** In-place Activation of a Calculator Server

## **OLE Concepts**

Without going too deep into OLE plumbing, you must get acquainted with some terms and concepts that rule the OLE protocol. Only then can you use the CDF efficiently.

**Components and Objects.** An OLE component or object can be anything that has a computer representation: a word processing document, a spreadsheet, a video clip, or even an application. This definition differs from the standard representation of an object in which an object contains data and functions to manipulate those data. Although the current implementation of OLE restricts the existence of any object to a single machine, a distributed version of OLE has already been demonstrated, which allows objects to span the enterprise network transparently.

**Containers and Servers.** OLE components that have other components embedded in them are container components, or for simplicity, *containers*. Components that can be embedded into containers are server components or, for simplicity, *servers*. Since embedded applications are called servers, containers are also called clients.

**Interfaces.** Interfaces define the protocol by which related functions of an object can be accessed. In contrast to C++ where a pointer to an object gives you access to every function exposed by the object, a pointer to an OLE object interface gives you access to only the functions the interface supports. In addition, OLE objects have multiple interfaces with relevant functions for each.

**Structured Storage.** To ensure the storage of a compound document, OLE-structured storage provides a file system within a file. In short, you can picture the OLE-structured storage as a DOS file system. Like the DOS file system, OLE-structured storage supports files and directories. Files can also be moved or copied. Data can be stored on disk, RAM, or CD ROM but also on other non native media as long as the specific methods for supporting those media are provided in OLE-structured storage.

**Uniform Data Transfer.** Uniform Data Transfer is a generalized intercomponent data transfer mechanism that rules data exchanges between servers and clients. In short, Uniform Data Transfer is OLE's equivalent of DDE and the clipboard cut-and-paste functionality all bundled into one. With this mechanism, data can be exchanged in multiple ways: using the clipboard, using the drag-and-drop metaphor, by links, or by in-place activation. Uniform Data Transfer supports its own notification when data has changed and includes a negotiation protocol to decide in which format the data are exchanged. Uniform Data Transfer supports the possibility that only a pointer or

handle on that data is transferred when the amount of data is large. This prevents the server from reading all the data before passing it to the client.

**Automation and Scripting Services.** Automation and Circulating Services define a set of interfaces which allows server components to be controlled by automation clients. In this case the client components are called automation controllers. Automation allows the controller to drive an automation object (server) through a set of predefined rules by using a scripting language. For example a Visual Basic application can control Excel and make it part of the Visual Basic application. The magic behind this ability is based on OLE's dynamic invocation features called *dispatchable interface*. When a client invokes a method or manipulates a property of the server, OLE's dynamic dispatch relies on a late-binding mechanism to resolve which method to call at run time.

**Linking and Embedding.** Linking and embedding allows a component client to tie to a component server. The tie can either be a link, in which the actual data carried in by the server resides outside of the client, or an inclusion, in which the data is embedded in the client component. OLE 2 provides a persistent naming service called *monikers* to gain access to the linked data. The moniker objects act as a persistent alias name for other objects and are used to insulate clients from a container. Monikers can provide aliases for all kind of objects from distributed file names to database queries or range of spreadsheet cells. As mentioned, OLE 2 supports in-place and out-place activation. For example, let us suppose that you embed an Excel 5 spreadsheet in a Word 6 document. If you double-click the spreadsheet, Excel activates (in-place activation) within Word and the Excel menu and toolbar replace the Word menu and toolbar. Clicking on a non-spreadsheet part of your Word document brings back the menu and toolbar of Word. If you now link your spreadsheet to your Word document and click on the spreadsheet, the Excel application is started, as in OLE 1, in a separate window (out-place activation).

When an OLE server is embedded, but not in use, it is inactive. When a server is inactive, a snapshot of its model data is used to indicate that the server is present. This inactive representation is referred to as the *metafile*. When creating an OLE server using the CDF, you need to draw its metafile as described under "Views" on page 461.

## The Compound Document Framework

The CDF helps you to create OLE-enabled applications that can function as containers or servers. With the framework you can save lot of time in developing such applications since the major code has already

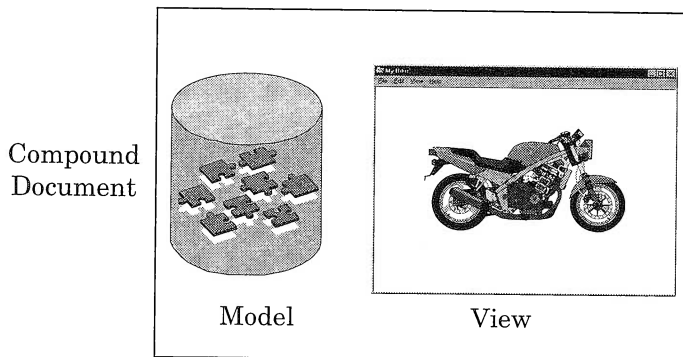
been developed just for you! In other words, instead of developing your applications from scratch, you need only customize the code provided by the framework.

Building an OLE application with the framework is simple as 1,2 3: first you define your model, the data it holds, and the storage strategy you want to use, then you define the representation of your data with a *view*, and finally you build a component *stationery* and its *GUI bundle* that define your application interface and glue all the pieces together.

In the followings paragraphs, we introduce you to each of the components you need to build your OLE application.

## Models and Views

Framework applications rely heavily on the model/view paradigm, which, should sound familiar to you since you have been playing with the Visual Builder for the major part of this book. An OLE application built with the framework has both a model that holds its data and a view that presents those data visually (Figure 161).



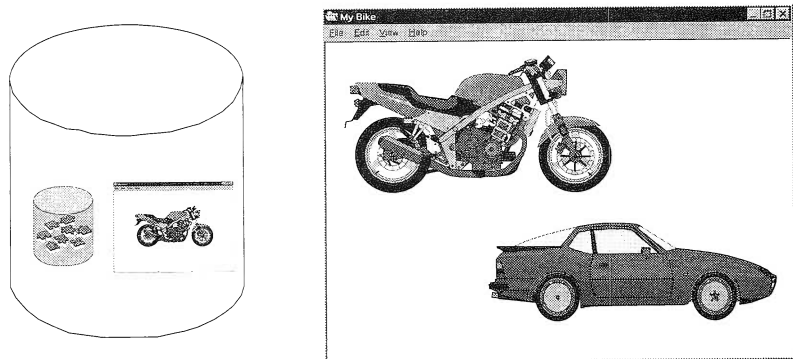
**Figure 161.** Model and View of an OLE Application

### Models

The model holds the application data and provides interfaces to modify the data it contains. Of course, the data can be of any type, such as picture, sound file, a graphic, or video. In order to build your model, you are provided with two model classes that your application must inherit from:

- ❑ **IModel** class provides models for server components. Servers that derive from IModel can be embedded but cannot have objects embedded in them. For example, paintbrush applications are typically servers since pictures can be inserted in word processors and spreadsheets, but objects are never inserted inside a paintbrush picture.
- ❑ **IEmbedderModel** class provides models for container components. Because IEmbedderModel is a subclass of IModel, containers that derive from IEmbedderModel have all the functionality of server components and can have other components embedded in them.

Embedder models manage the embedding of components through a list of **IEmbeddedComponent** objects and provide services to them including rendering them to the screen. In this way, when a component is embedded in a container component, it is actually stored in the embedder model of the container. Containers can be embedded themselves in other containers building up a container hierarchy; the highest level container is called the *root container component*. The root container component stores the data of all embedded containers and servers in its structured storage as shown in Figure 162.



**Figure 162.** Component Embedded within a Container Embedder Model

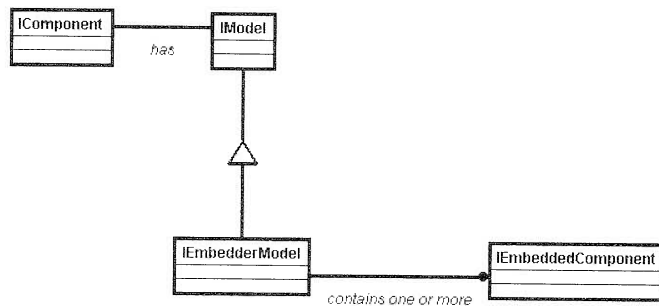
The term *embedding* is generally used to refer to both embedding and linking, because linking operations have essentially the same characteristics as embedding operations. The key exception is that linked data is not stored in the container's structured storage file. Instead, the container's structured-storage file maintains a reference to the path where the servers's file is stored. This way, the linked data can be shared with other linked applications.



Finally, **IComponent** class provides an interface to the embedded components to allow the model of the container to interact with its containing components. The IComponent class acts as a proxy for an OLE component. Four key objects are involved in the embedding protocols:

- ❑ IModel
- ❑ IEmbedderModel
- ❑ IEmbeddedComponent
- ❑ IComponent

The collaboration among these four classes is summarized in Figure 163.



**Figure 163.** Key Objects of an Embedding Relationship

Since a compound document may contain different types of data, the embedder model for the root container component requires a specific mechanism to store its own native data along with the native data of any embedded components in a single hierarchically organized file.

As mentioned, OLE provides this mechanism as structured storage. OLE structured storage files are referred to as *compound files*. OLE compound files use storage entities and streams to build up the hierarchy within a file and provide the ability to store data of different types. In a conventional file system such as the one provided by DOS, storage entities can be compared to directories and streams to files. Together, storage entities and streams provide a file system within a file.

**Storing Model Data.** Data from the model must be saved on storage media to ensure that it survives the closing of components. For this reason, model data are referred to as *persistent* data. Data are usually kept in the model, but you can choose to keep them in some other object, such as a database, and maintain references to them in the model.

To provide you with a mechanism that move components in and out of storage, the CDF defines an **IDocumentStorage** abstract class. The document storage object is created and owned by the component (that is, an object of class **IComponent**).

Two concrete derived classes of **IDocumentStorage** are defined to let you choose the type of storage you need for your server:

- ☐ **IFlatFileStorage**
- ☐ **IStructuredStorage**

Servers can either save their data into a flat file or into structured storage. Since servers do not handle embedded components, they use flat file storage by default.

Containers must save their data into structured storage in order to save the data of their embedded components.

**Streaming Model Data.** When a document is saved, the framework creates a stream using the storage. This stream object is passed to the stream-out operator of a component's model. Similarly, the stream object is passed to the stream-in operator when a document is opened.

To build your server model you basically need to:

- ☐ Decide the data you want to manage and provide public member functions to access and modify it.
- ☐ Send notifications to the associated view, if necessary, when data changes.
- ☐ Implement a strategy regarding how your data are to be persistent (database storage, flat file storage, other)

## Views

A model needs a view to visually represent its data to the end-user. Views are defined by subclassing the **IView** class. By creating a class which derives from **IView**, you can control the display of data contained in the model and the user interactions with the data. The derived class you create is one of the template parameters (along with the model) used to create your component stationery.

Whenever a “contents” presentation of the view is required, the framework call the **IView::draw** member function. This is done primarily in response to paint messages but also to provide the presentation used in OLE caches (metafile) when the server that resides in the container is not active. The draw function first calls the **drawContents** function to allow native data to be drawn. Then, if the component is a container, it iterates through all embedded objects and calls on them to draw recursively.

Your main task is to override the `IView::drawContents` in order to have the data shown on the screen. Basically two representations of your data must be drawn on the screen:

- ❑ The representation of the data, in the container, when the server is in-place activated.
- ❑ The representation of the data, in the container, when the server is not running or when it is running out of place. In this case you draw in a specific presentation space and call the metafile, which provides the cached presentation used by the OLE container to represent your server.

Models and views interact with each other by using a notification event that are based on the Visual Builder notification framework (see Chapter 10, “More about Visual Builder...,” on page 353). When the model data changes, the setter methods notify the view by using the *IModel::notifyOfChange* member function. The view can check the notification event and react accordingly in updating the representation data on the screen. For this purpose, you must override the *IView::handleNotification* member function. Of course you can add other event handlers if you want your view to react to user interactions such as clicking a push button or double-clicking a list box entry.

### ***Component Stationery and GUI Bundle***

When your view and your model are implemented, you need to glue them together to make your server a self-contained component. The component stationery, derived from **IComponentStationery** is used for this purpose. The component stationery, also referred to as the *stationery*, also provides each part of your application a way to access any other part. For example, through the stationery, the model can easily access the view and the other way round.

The stationery also registers and makes available certain OLE-specific clipboard formats, and groups them into format sets that are used for data transfer between applications.

Basically, you can create your application's stationery in three different ways:

- ❑ Derive directly from the `IComponentStationery` abstract class and provide an implementation for the pure virtual functions: `IComponentStationery::createModel` and `IComponentStationery::createView`.
- ❑ Use the template class `IComponentStationeryFor` and use its implementation for the two pure virtual functions.
- ❑ Derive from `IComponentStationeryFor` and override its `createView` and `createModel` virtual functions to enhance your application.

When the stationery is created, it creates a GUI bundle object (instance of the **IGUIBundle** class) which owns and controls the model and view of your application. The GUI bundle gathers run-time information for your application and provides it with some basic user interface controls such as a frame window and a toolbar. It also supplies the menu items needed to support the OLE protocol.

For simple applications, the default behavior of **IGUIBundle** should be satisfactory. If needed, you can derive your class from **IGUIBundle** and provide your own behavior to customize your application frame window, its menu, or its toolbar.

### ***CLSID and System Registration Database***

Every CDF application requires a class identifier (CLSID) to be identified uniquely. The CLSID is stored as a string in your application's resource file and is identified by **IC\_CDF\_CLSID**. The first time you run your application (either server or container), the CLSID is automatically registered in the system registration database, also referred to as the *registry*, under the key **HKEY\_CLASSES\_ROOT**. You can easily access this key using the **regedit** utility under Windows 95 or the **regkey** utility provided with the NT resource kit for Windows NT.

The registry is used extensively by OLE. For example, when an end-user selects **Insert Object** from a container component, the list of objects that appears is built up using information in the registry.

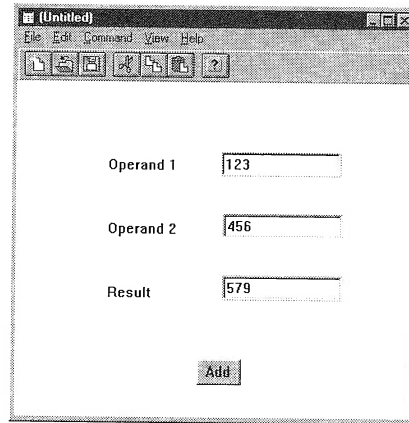
CLSIDs are created using series of random numbers combined with the current date and time of the system and the system's hardware characteristics (such as your network adapter physical address) to make sure the number is globally unique. You can generate CLSID for your application in two main ways:

- You can use the **UUIDGEN.EXE** utility located in the **\IBMCPW\BIN** directory to generate the CLSID of your application and paste the generated string in your application's resource file.
- You can run Project Smarts (as described later in this chapter) to create a skeleton framework application for you, including its CLSID.

Now that you are acquainted with the CDF, you are ready to develop your first application. To mirror the structure of Chapter 12, you develop a simple component server that represents a calculator. For this first example, we show you how to build your server without using the Visual Builder. In the next example, we reuse "ASimulMortgageView" and "AMortgageCalculator" to illustrate how you can take advantage of the Visual Builder and create your view and your model more quickly.

## Building a Component Server

The component server you build is a simple calculator that takes two operands, adds them, and returns a result. Once the server is developed, you can embed it in any OLE container and activate it within the container as you would using a standard calculator program (Figure 164).

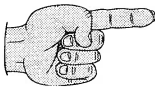


**Figure 164.** The Calculator Component Server

Basically, you build a server in three steps:

1. You create a model for your server.
2. You create a view for your server.
3. You create a component stationery based on your model and view.

### Read This



The samples built in this chapter are constructed so that the Open Class Libraries are dynamically linked to your application. When running your server, you must ensure that the directory containing the Open Class Libraries DLLs (`\IBMCPW\BIN`) is in your Windows/NT System path.

## Creating Your Project with WorkFrame

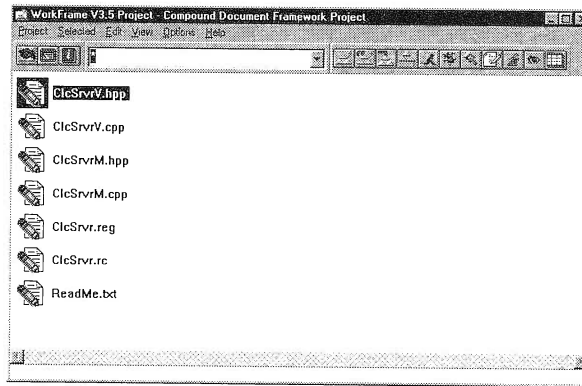
To create a project skeleton for your calculator server, you use WorkFrame as follows:

1. In the IBM VisualAge for C++ for Windows container, double-click the **WorkFrame IDE** icon
2. Click the **Create new project** push button. The Project Smarts notebook is displayed on the screen.
3. Scroll down the Project types list, select the **Compound Document Framework** type and click the **Next** push button.
4. In the Location page, fill out the entry fields as follow:

<b>Project Title</b>	Calculator Server
<b>Source File Directory</b>	D:\IBMCPW\WORKING\OLECALC
<b>Where to Create the Project Files</b>	D:\IBMCPW\WORKING
<b>Project File Name</b>	OLECALC

5. Click the **Next** push button
6. In the Component page, select **Server** as the Component type and fill in the Component name entry field with **CalcServer**. Click the **Next** push button.
7. In the Application Name page, **CalcServer** is selected as the Application Short Name. This name is used to register the component in the Windows registry and appears also in the context menu for any objects the component can edit. Likewise, **CalcServer Document** is automatically selected as the Application Long Name (this long name is displayed in the application's title bar). Click the **Next** push button
8. In the Model Class page, **CalcServerModel** is automatically selected as your model class name. Click the **Next** push button.
9. In the View Class page, **CalcServerView** is selected as your view class name. Click the **Next** push button.
10. In the File Extension page, the **cal** extension is automatically proposed for all files that are saved and loaded by your calculator component. Click the **Next** push button.
11. In the Prologue page, fill in the different entry fields to your liking and click the **Done** push button.

A project is created by Project Smarts and filled already with various files (Figure 165).

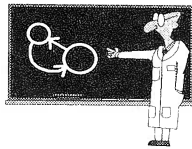


**Figure 165.** Calculator Server Project

Seven files are generated for your convenience:

- ☐ **ClcSrvrV.hpp**: header file for the CalcServerView class which derives from the IView class.
- ☐ **ClcSrvrV.cpp**: implementation of the CalcServerView class
- ☐ **ClcSrvrM.hpp**: header file for the CalcServerModel class which derives from the IModel class.
- ☐ **ClcSrvrM.cpp**: implementation of the CalcServerModel-derived class
- ☐ **ClcSrvr.reg**: registration information file for your application
- ☐ **ClcSrvr.rc**: resource file of your application
- ☐ **ReadMe.txt**: more explanations!

#### Technical Information



The registration file provided with your project is intended to be used with a setup program to install your application. Your application server must be registered explicitly before it can be embedded in another component. If such a program is not provided with your application, the registration information is added to the Windows registry the first time your application runs by calling `InitInstance` which in turn calls the methods: `CWinApp::RegisterShellFileTypes` and `COleObjectFactory::UpdateRegistryAll`. You can also register your server by double-clicking on the registration file.

## Creating a Server Model

To create the server model of our calculator, you must decide which data, and functions to manipulate the data, are required for the calculator. In this sample application the decision is easy. You need three attributes ***operand1***, ***operand2***, ***result***, their get and set accessors, and a member function that calculates the addition of two numbers. Let us choose an `IString` representation for every number and for the return value of the addition member function. You also need to decide how the persistence of your data is implemented. To make things simple, let us decide that you use the streaming framework provided by CDF to save the value of your three attributes.

You can now refine your class `CalculatorModel` as follows:

1. Declare the model's data along with the setter/getter member functions and the add member function.
2. Define the constructor, destructor, and copy constructor.
3. Define the streaming for the model.

In the `ClcSrvrM.hpp` header file, we outline the code that you must add or customize to build the calculator model:

```

01
02 #ifndef _CLCSRVRM_
03 #define _CLCSRVRM_
04
05 /*
06  *COMPONENT_NAME: ClcSrvrM.hpp
07  *
08  *FUNCTIONS: mainly setter and getter functions for the attributes
09  *
10  *PURPOSE:
11  */
12
13 #include <istring.hpp>
14 #include <imodel.hpp>
15
16 //-----
17 //  CalcServerModel
18 //-----
19
20 class CalcServerModel : public IModel
21 {
22 public:
23     TypeExtensionDeclarationsMacro(CalcServerModel)
24
25     CalcServerModel();
26     virtual ~CalcServerModel();
27
28     virtual IBaseStream& operator>>= ( IBaseStream& toWhere ) const;

```



```
28     virtual IBaseStream&    operator<=<= ( IBaseStream& fromWhere );
29
30     //////////////////////////////////////
31     //  ADD your public member functions here
32     //////////////////////////////////////
33
34     virtual IString getOperand1() const;
35     virtual Boolean setOperand1(const IString&);
36
37     virtual IString getOperand2() const;
38     virtual Boolean setOperand2(const IString&);
39
40     virtual IString getResult() const;
41     virtual Boolean setResult(const IString&);
42
43     virtual IString add();
44
45     protected:
46         CalcServerModel( const CalcServerModel& copy );
47     private:
48         // Hide assignment operator
49         CalcServerModel& operator=( const CalcServerModel& copy );
50         enum              { kOriginalVersion };
51
52         //////////////////////////////////////
53         //  ADD your data members here
54         //////////////////////////////////////
55
56         IString operand1;
57         IString operand2;
58         IString result;
59
60     };
61
62
63 #endif // _CLCSRVRM_
```

Each line in the header file is described briefly as follows (for more information, consult the VisualAge for C++ Open Class Library User's Guide):

1. Lines 12 and 13 include the needed header files for manipulating IString objects and for using the framework model.
2. Line 19 defines the CalcServerModel class as a public derived class of IModel.

3. Line 22 declares a macro for the Extended Type system which provides run-time type identification (RTTI) and other functionality required by all framework models, such as the `dynamic_cast` operator.
4. Lines 24 and 25 declare the model default constructor and the virtual destructor.
5. Lines 27 and 28 declare streaming operators used to save and load the three attributes defined for the model.
6. Lines 34 through 43 declare the attributes get and set accessors and the add member function.
7. Line 45 declares a protected copy constructor which is mandated by the framework.
8. Line 49 declares the assignment operator needed also by the framework.
9. Line 50 declares an enum to create the `kOriginalVersion` number of the document. This version number is used during streaming-in and streaming-out to support the application evolution over the time and to let different versions of a document be readable.
10. Lines 56 through 58 define the model data.

In the following sections, we provide a more detailed explanation of the way you should write the code for your model.

### ***Defining Data Members and Accessors***

You structure your model's data according to the requirements of your application:

#### **In the .hpp file:**

1. Declare ***operand1***, ***operand2***, and ***result*** as `IString` attributes in the private portion of your class definition as follows:

```
IString operand1;
IString operand2;
IString result;
```

2. Declare the corresponding get and set accessors in the public portion of your class definition as follows:

```
virtual IString getOperand1();
virtual Boolean setOperand1(const IString&);
virtual IString getOperand2();
virtual Boolean setOperand2(const IString&);
virtual IString getResult();
virtual Boolean setResult(const IString&);
```

3. Declare the add member function in the public portion of your class definition as follows:

```
virtual IString add();
```

**In the .cpp file:**

1. Implement the set and get accessors for each attribute as follows:

```
IString CalcServerModel::getOperand1() const
{
    return operand1;
}
Boolean CalcServerModel::setOperand1(const IString& str)
{
    operand1 = str;
}

IString CalcServerModel::getOperand2() const
{
    return operand2;
}
Boolean CalcServerModel::setOperand2(const IString& str)
{
    operand2 = str;
}

IString CalcServerModel::getResult() const
{
    return result;
}
Boolean CalcServerModel::setResult(const IString& str)
{
    result = str;
}
```

2. Implement the add member function as follows:

```
IString CalcServerModel::add()
{
    return setResult(IString(operand1.asDouble() +
                             operand2.asDouble()));
}
```

## Construction and Destruction of the Model

Once your model's data are defined, you refine the model default constructor, which is declared as public in the model header file. In this example, you do not need to refine the default destructor that ProjectSmarts generates for you:

1. Refine the default constructor in `CalcServerModel.cpp`:

```
CalcServerModel::CalcServerModel() : IModel()
{
    operand1 = IApplication::current().userResourceLibrary()
                                   .loadString(STR_EMPTY);
    operand2 = IApplication::current().userResourceLibrary()
                                   .loadString(STR_EMPTY);
    result    = IApplication::current().userResourceLibrary()
                                   .loadString(STR_EMPTY);
}
```

The code sets the contents of each attribute with their corresponding initial string. These strings are added to the resource file using the resource editor (“Creating the Resources” on page 476).

2. The default destructor is already implemented in `CalcServerModel.cpp` as shown:

```
CalcServerModel::~CalcServerModel()
{
}
```

Make sure that the default destructor is declared as **virtual** in the model header file to ensure that the correct destructor is called whenever your object is deleted by means of a base class pointer to your model object.

## Defining the Copy Constructor

In order to compile your program, the framework Extended Type system requires that you implement a copy constructor for your model class. Luckily, Project Smarts generates such a copy constructor for you and declares it in the private section of your model's header file. All you need to do is initialize the data members as follows:

```
CalcServerModel::CalcServerModel( const CalcServerModel& copy)
: IModel( copy )
{
    operand1 = operand2 = result = Istring();
}
```

In the current release, copying of models is not supported by the CDF and would generate a run-time error.

### ***Streaming the Model***

As mentioned, the framework provides you with functions that are used to stream your data out from the model (saved to a file) or stream the data into the model (loaded from a file). These functions are called by the framework when the user selects specific menu items such as **File → Open or File → Save**.

To enable streaming for your model's data, you must in turn provide your model class with:

- ❑ Specific macros for the Extended Type system
- ❑ An assignment operator, copy constructor, and default constructor
- ❑ A version number (optional but recommended)
- ❑ The stream-out (>>=) and stream-in (<<=) operators

When you create the server project with Project Smarts, all the necessary code is already generated for you in the model class header and implementation files. You just need to refine the streaming operators to reflect the model's data you want to make persistent.

Refine the stream-out operators as shown (the lines of code that you must add are in bold type):

```
IBaseStream& CalServerModel::operator>>=( IBaseStream& toWhere ) const
{
    writeVersion( toWhere );
    IModel::operator>>=( toWhere );

    //////////////////////////////////////
    // Add stream out of data members here //
    //////////////////////////////////////
    operand1 >>= toWhere;
    operand2 >>= toWhere;
    result >>= toWhere;

    return toWhere;
}
```

First, the stream-out operator writes the document version to the stream using the `::writeVersion` global function. Then it calls the base class stream-out function using the `IModel::operator>>=`. Finally, the model's data are written to the stream. Similarly, you refine the stream-in operator as follows:

```
IBaseStream& CalServerModel::operator<<=( IBaseStream& fromWhere)
{
    switch( readVersion( fromWhere ) ) {
```

```

        case kOriginalVersion:
            IModel::operator<<= ( fromWhere );

            //////////////////////////////////////
            // Add stream out of data memebers here //
            //////////////////////////////////////
            operand1 <<= fromWhere;
            operand2 <<= fromWhere;
            result <<= fromWhere;
            break;
        default:
            ITHROWLIBRARYERROR(IC_STREAM_VERSION_UNSUPPORTED,
                               IBaseErrorInfo::invalidRequest,
                               IException::recoverable);
    }
    return fromWhere;
}

```

The stream-out operator first checks the version number of the data and, if it matches the original version, then streams the data in. Notice that Project Smarts generates a switch statement that allows you to implement your own code for supporting multiple version of your model's data.

The streaming operators are called by the framework, passing a stream object (IBaseStream subclass) of the storage being used. Flat file storage is used as default storage for the server and is suitable in most cases. However, if you need structured storage you can override the IComponent::isStructuredStorage function to return the value *TRUE* (consult the IBM Open Class Library Reference for more information).

## Creating a Server View

After defining the model class for your server, you must define a view class to visualize your model's data. To do so, you must carry out the following steps:

1. Determine the layout of your view, the controls used and their position. For the calculator, use one push button, three entry fields, and three static texts as depicted in Figure 164.
2. Define a constructor and destructor for your view.
3. Define a downcasting function for accessing your model's data and update the view accordingly.
4. Set up an initialization routine for your view to display the model with its initial data.
5. Draw the contents of your view using the User Interface Class Library each time the model's data change.

Again, Project Smarts comes to give you a hand in generating most of the code necessary to build the view.

Let us review first the ClcSrvrV.hpp header file generated by Project Smarts and the code you must add (in bold type):

```
01 #ifndef _CLCSRVRV_
02 #define _CLCSRVRV_
03 /*
04  *COMPONENT_NAME: ClcSrvrV.hpp
05  *
06  *FUNCTIONS:
07  *
08  *PURPOSE:
09  */
10
11 #include <iview.hpp>
12 #include <icmdhdr.hpp>
13 #include <ipushbut.hpp>
14 #include <ientryfd.hpp>
15 #include <istattxt.hpp>
16
17 //-----
18 // CalcServerView
19 //-----
20 class CalcServerModel;
21 class CalcServerView : public IView
22 {
23     public: virtual ~CalcServerView();
24             CalcServerView( IGUIBundle& bundle );
25             virtual void initialize();
26             virtual void finalize();
27             virtual Boolean handleCommand( ICommandEvent& cmdEvent );
28     //////////////////////////////////
29     // ADD your public member function here
30     //////////////////////////////////
31     virtual CalcServerModel* getModelPointer() const;
32     protected:
33             virtual void drawContents( IPresSpaceHandle& ps,
34                                     const IRectangle&, Boolean );
35     private:
36             // Hide assignment operator
37             CalcServerView& operator=( const CalcServerView& copy );
38             // Command Connection for this View
39             ICommandConnectionTo<CalcServerView> fCommandHandler;
40     //////////////////////////////////
41     // ADD handlers, controls, and other data here
42     //////////////////////////////////
43     IPushButton pushButtonAdd;
44     IStaticText staticText0perand1;
```

```

45  IStaticText staticTextOperand2;
47  IStaticText staticTextResult;
48  IEntryField entryFieldOperand1;
48  IEntryField entryFieldOperand2;
49  IEntryField entryFieldResult;
50  };
51  #endif // _CLCSRVRV_

```

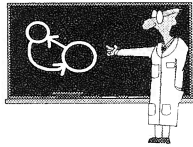
Each line in the view header file is described briefly as follows:

1. Line 11 includes the `IView` base class header.
2. Line 12 includes the  `ICommandHandler` and  `ICommandConnectionTo` base classes header that are needed to handle the events when a user interacts with the server.
3. Lines 13 through 14 include the header files necessary to implement the view of the server using the push button, static text and entry field controls from the User Interface Class Library.
4. Line 20 forward declares the `CalcServerModel` that you have already defined. Forward declaration is needed since you declare in Line 31 a *getModelPointer* member function which returns a pointer to the current model and lets the view access your model's data.
5. Line 21 declares the `CalcServerView` as a public subclass of `IView`.
6. Lines 23 and 24, respectively, declare the default destructor and constructor of your view. Notice that the destructor must be declared *virtual*.
7. Line 25 declares the member function *initialize* called by the framework to initialize the view. Its default implementation starts the message handlers for painting, mouse event and notifications. You override this function in the .cpp file to perform some post-construction tasks required in the view.
8. Line 26 declares the member function *finalize*, which does essentially the opposite of the *initialize* function in closing down the view. If you declare and implement the *initialize* function, you must also declare and implement the *finalize* function, which is called by the framework in order to shut the view down properly.
9. Line 27 declares a handler function for command events, while Line 39 defines, in the private section, the actual handler. This handler can be used to control user interaction with the server.



10. Line 31 declares a convenience function which returns a pointer to the model using dynamic casting (explained later in this chapter).
11. Line 33 declares the *drawContents* function that you override in the .cpp file to provide your own view contents and the aspect of your model's metafile.
12. Line 37 declares in the private portion the assignment operator needed by CDF.
13. Lines 43 through 48 define the user interface objects you need to build your view.

#### Technical Information



The `ICommandHandlerConnectionTo` class is a template class, derived from `ICommandHandler`, which processes application and system command events and routes them to the template argument class (`CalcServerView` in your case). This class allows you to process command events in objects that do not derive from `ICommandHandler` without having to manually derive from `ICommandHandler`, override the `command` and `systemCommand` functions, and pass the events to the object. For more information, refer to the IBM Open Class Library Reference Volume II.

In the following sections we provide a more detailed explanation of the way you should customize your resource scripts file and modify the code for your view.

### Creating the Resources

Resources are binary data that are linked to an .exe, .dll or a binary file to define user interface components such as menu, cursor, strings or icons. Although those resources can be coded directly into your program, it is always a good design to define them outside your application, in a resource file, to let you easily change the user interface of your application and support different national languages.

You create and edit resources in resource script files that contain one or more sets of ASCII data, called *resource scripts*, from which the resource compiler generates binary resources. You can create and edit resource script files directly, hacking the scripts using an ASCII editor, or you can use Resource Workshop to generate the resource script files for you.

In addition, a resource script contains one or more identifiers. An identifier is a unique name or number that you can use in your code to refer to a resource. Those identifiers can be created automatically in Resource Workshop when you add new resources or you can create them manually (for more information consult the VisualAge for C++ for Windows User's Guide).

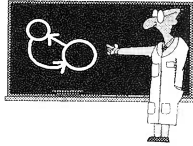
For your calculator server, Project Smarts has already generated a resource file, `ClcSrvr.rc`, which contains the

- ☐ Server menu definition
- ☐ Server pop-up menus
- ☐ Menu and pop-up menu accelerators
- ☐ Autoregistration information strings
- ☐ Dynamic menu items strings

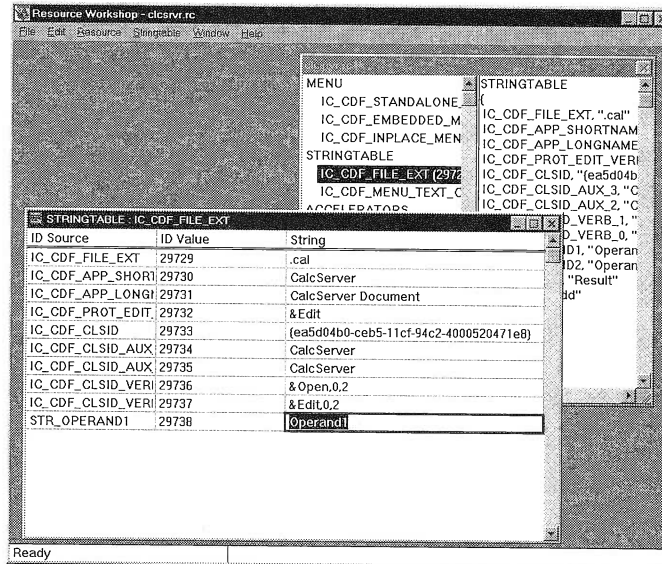
In order to build the view depicted in Figure 164, you add your own resources string resources and their identifiers to the resource file as follows:

1. In the Workframe project, double-click the resource file to open the resource editor.
2. From the menu bar, select **File** → **Preferences** and verify that the *Generate identifiers automatically* check box is *not selected*. This way, you can select manually the identifier number of your resource each time you add a new resource. Close the dialog box by clicking the **OK** push button.
3. From the menu bar select **Resource** → **New...**, a dialog box is displayed. In this dialog box, select **STRINGTABLE** as the Resource type. You are about to add string resources and their identifiers to the actual resource file generated by Project Smarts. Click the **OK** push button; a three-column table then displays on the screen.
4. In the first row of the first column already highlighted, type in **STR\_OPERAND1** as the ID Source of your first string. Then hit the **Tab** key. A message box appears and prompts you to validate the creation of the new identifier STR\_OPERAND1. Click the **Yes** push button to display the *New identifier* dialog box. Enter **10000** in the Value entry field and select `clcsrvr.rc` in the File drop-down list box (when you save the resource project, the STR\_OPERAND1 and its identifier are added to the current resource script file). Click the **OK** push button.
5. The last column of the STRINGTABLE table is highlighted. Enter the string **Operand1** (Figure 166) and press the Insert key. You are ready to enter the next string resource.

## Technical Information



It is good practice to separate your identifier definitions from your resource scripts file. Usually, the identifiers are defined in a separate header file with the `.h` extension. This header file is then included in the `.cpp` file that uses the identifiers.



**Figure 166.** STRINGTABLE Window

Repeat these steps to insert the rest of the string resources as shown in Table 74.

**Table 73.** (Part 1 of 2) String Table Resources for CalcServerView

ID Source	ID Value	String
STR_OPERAND1	10000	Operand1
STR_OPERAND2	10001	Operand2
STR_RESULT	10002	Result
STR_ADD	10003	Add

<b>Table 73.</b> (Part 2 of 2) String Table Resources for CalcServerView		
<b>ID Source</b>	<b>ID Value</b>	<b>String</b>
STR_EMPTY	10004	Empty

To complete the modification of your resource scripts file, you need to add the identifiers of the different controls of your view:

1. From the Resource Workshop menu bar, select **Resource Identifier...** The Identifiers dialog box is displayed on the screen.
2. Click the **New** push button to add a new identifier to the resource scripts file.
3. Fill in the name entry field with **ID\_BUTTON**; fill in the value with **11000** and make sure **clcsrvr.rc** is selected in the File drop-down list box.
4. Click the **OK** push button to create the new identifier for the Add push button of your view.
5. Repeat Steps 1 through 3 to add the other identifiers and their corresponding value, as shown in Table 74.

<b>Table 74.</b> Resource Identifiers for CalcServerView		
<b>ID Source</b>	<b>ID Value</b>	<b>String</b>
ID_BUTTON	11000	ClcSrv.rc
ID_STATIC1	11001	ClcSrv.rc
ID_STATIC2	11002	ClcSrv.rc
ID_STATIC3	11003	ClcSrv.rc
ID_ENTRY1	11004	ClcSrv.rc
ID_ENTRY2	11005	ClcSrv.rc
ID_ENTRY3	11006	ClcSrv.rc

When all the identifiers are created, click the **Done** push button to close the Identifiers dialog box. Then, from the Resource Workshop menu bar, select **File** → **Save resource project** to save the changes in the resource scripts file.

Optionally, you can add your own icon and its identifier (IC\_CALC) as a new resource for your calculator server application (consult the VisualAge for C++ for Windows User's Guide, for more information about the resource editor).

## Creating the Constructor and Destructor

You use the User Interface Class Library from the Open Class Library to build your view and refine the constructor generated by Project Smarts in `ClcSrvrV.cpp` as follows (the lines you add are shown in bold type):

```
CalcServerView::CalcServerView( IGUIBundle& bundle ) :  
    IView( bundle ),  
    fCommandHandler( *this, handleCommand ),  
    pushButtonAdd(ID_BUTTON, this, this),  
    staticTextOperand1(ID_STATIC1, this, this),  
    staticTextOperand2(ID_STATIC2, this, this),  
    staticTextResult(ID_STATIC3, this, this),  
    entryFieldOperand1(ID_ENTRY1, this, this),  
    entryFieldOperand2(ID_ENTRY2, this, this),  
    entryFieldResult(ID_ENTRY3, this, this)  
{  
    bundle.framewindow().setIcon(IC_CALC); // Set the window icon  
}
```

The constructor calls the base class constructor *IView(bundle)* and then constructs the view with the different controls. The *IGUIBundle&* argument in the constructor links the view to the user interface elements provided by default by the framework such as the frame window and menu choices. Last, a call to the *IFrameWindow::setIcon* member function sets the icon of the server.

Notice that in order to manipulate your resource identifiers and the *IFrameWindow* class, you need to include the header files *ireslib.hpp* and *iframe.hpp* at the top of the *.cpp* file. Also, in order to implement the *getModelPointer* member function that you declare in the *.hpp* file, you need to include the model header, *ClcSrvrM.hpp* as shown in the following lines:

```
/*  
 *COMPONENT_NAME: ClcSrvrV.cpp  
 *  
 *FUNCTIONS:  
 *  
 *PURPOSE:  
 */  
  
#include <ireslib.hpp>  
#include <iframe.hpp>  
#include "ClcSrvrM.hpp"  
#include "ClcSrvrV.hpp"
```

The destructor for the view is already created by Project Smarts and does not require any modification.

## Downcasting the Model

To access your model's data from the view, you use the *CalcServerView::getModelPointer()* member function which returns a pointer to your model. This function uses a special mechanism, referred to as *dynamic casting* that we describe in the following sections.

**Static Casting.** The framework provides an *IView::model()* member functions which returns a pointer to the model. The pointer returned is a pointer to the *IModel* base class and not to your own model class *CalcServerModel*. Hence, whenever you need to access your model, you must cast the *IModel* pointer into a *CalcServerModel* pointer as follows:

```
CalcServerModel* pMyModel = (CalcServerModel*) model();
```

This operation is known as *static casting* and is safe for base class or derived class that inherits from only one class (single inheritance). For multiple inheritance, this casting can lead to errors. For this reason, the framework's Extended Type system provides dynamic casting in returning the correct pointer of all type of classes.

**Dynamic Casting.** Dynamic casting is provided throughout the global function *IMetaTypeInfo::dynamicCastTo(AType \*&, BType\*)* that takes a pointer of *BType* and tries to downcast it dynamically to a pointer of *AType*. The cast fails if the derived class *AType* does not derive from the class *BType*. In this case the resulting pointer is set to *NULL*. You use the dynamic casting to implement *getModelPointer* as follows:

```
CalcServerModel* CalcServerView::getModelPointer() const
// Get the pointer to the model
{
    CalcServerModel* theModel = NULL;
    ::dynamicCastTo( theModel, model() ); // downcast the model pointer
    return theModel;
}
```

## Initializing the View

The *initialize* function allows you to perform post-processing to initialize your view. Use the *getModelPointer* member function to access your model's data and update the view accordingly:

```
void CalcServerView::initialize()
{
    IView::initialize();

    //////////////////////////////////////
    // ADD initialization code here
    //////////////////////////////////////
```

```
staticTextOperand1.moveSizeTo(IRectangle(IPoint(10,10), IPoint(100 ,40)));
entryFieldOperand1.moveSizeTo(IRectangle(IPoint(200,10), IPoint(300 ,40)));
staticTextOperand2.moveSizeTo(IRectangle(IPoint(10,50), IPoint(100 ,80)));
entryFieldOperand2.moveSizeTo(IRectangle(IPoint(200,50), IPoint(300 ,80)));
staticTextResult.moveSizeTo(IRectangle(IPoint(10,90), IPoint(100 ,120)));
entryFieldResult.moveSizeTo(IRectangle(IPoint(200,90), IPoint(300 ,120)));
pushButtonAdd.moveSizeTo(IRectangle(IPoint(100, 120), IPoint(150, 150)));

CalcServerModel* myModel = getModelPointer();
assert ( myModel != NULL );
staticTextOperand1.setText(STR_OPERAND1);
staticTextOperand2.setText(STR_OPERAND2);
staticTextResult.setText(STR_RESULT);
entryFieldOperand1.setText(myModel->getOperand1());
entryFieldOperand2.setText(myModel->getOperand2());
entryFieldResult.setText(myModel->getResult());
pushButtonAdd.setText(STR_ADD);
fCommandHandler.handleEventsFor( this );
}
```

First, the *initialize* function calls the *initialize* member function of its base class. Then the controls are laid out and sized on the view using the *moveSizeTo* member function. The *getModelPointer* function retrieves a pointer on the *CalcServerModel* that is used to update the contents of the entry fields with the model's data and the static text are initialized with the string resources that were defined previously using Resource Workshop.

### ***Drawing the Contents of the View***

Whenever a “contents” presentation of the view is required, the framework calls the *IView::draw* member function. This is primarily in response to paint messages, but the *draw* member function also provides the model with a presentation used in an OLE cache, also known as *metafile*. This metafile provides the cached presentation that OLE containers use to display the view when the embedded server component is not active or when it is running out of place. Without metafile, the server would appear as an empty area in the container. In turn, the *draw* member function calls the *IView::drawContents* member function to draw the native data in the view.

The framework provides three parameters to the *drawContents* member function:

- ❑ A presentation space handle (*IPresSpaceHandle*) into which you can draw
- ❑ An invalid rectangle (*IRectangle*) that can be used to optimize drawing

- A boolean which is TRUE if the server is active or not, which means whether you draw to a metafile or to the screen

The default implementation of `IView::drawContents` does nothing. Your job is to override this member function in your view class to provide the appropriate rendering for your metafile and speed up the drawing process if needed.

Refine the `drawContents` member function of `CalcServerView` in the `.cpp` file as follows:

```

01 void CalcServerView::drawContents( IPresSpaceHandle& ps,
02                                   const IRectangle&,
03                                   Boolean metaFile)
04 {
05     //////////////////////////////////////
06     // ADD Drawing of your view here
07     //////////////////////////////////////
08     CalcServerModel* myModel = getModelPointer();
09     assert(myModel != NULL);
10
11     if (metaFile)        // Place the embedded image code here !!!!
12     {   IString str1, str2, str3, ef1, ef2, ef3;
13         IGraphicContext ctxt (ps );
14
15         ctxt.setFillColor(IColor::white);
16         IRectangle myRect(IRectangle(IPoint(10,10),IPoint(220,110)));
17         str1 = IApplication::current().userResourceLibrary()
18               .loadString(STR_OPERAND1);
19         str2 = IApplication::current().userResourceLibrary()
20               .loadString(STR_OPERAND2);
21         str3 = IApplication::current().userResourceLibrary()
22               .loadString(STR_RESULT);
23
24         IGString static1(str1,IPoint(20,20));
25         IGString static2(str2,IPoint(20,50));
26         IGString static3(str3,IPoint(20,80));
27
28         if (myModel->getOperand1().length() > 0)
29             ef1 = myModel->getOperand1();
30         else
31             ef1 = IApplication::current().userResourceLibrary()
32                   .loadString(STR_EMPTY);
33         if (myModel->getOperand2().length() > 0)
34             ef2 = myModel->getOperand2();
35         else
36             ef2 = IApplication::current().userResourceLibrary()
37                   .loadString(STR_EMPTY);
38         if (myModel->getResult().length() > 0)
39             ef3 = myModel->getResult();
40         else
41             ef3 = IApplication::current().userResourceLibrary()
42                   .loadString(STR_EMPTY);

```



```
43
44     IGString gef1(ef1,IPoint(150,20));
45     IGString gef2(ef2,IPoint(150,50));
46     IGString gef3(ef3,IPoint(150,80));
47
48     myRect.drawOn(ctxt);
49     static1.drawOn(ctxt);
50     static2.drawOn(ctxt);
51     static3.drawOn(ctxt);
52     gef1.drawOn(ctxt);
53     gef2.drawOn(ctxt);
54     gef3.drawOn(ctxt);
55 }
56 else
57 {
58     entryFieldOperand1.setText(myModel->getOperand1());
59     entryFieldOperand2.setText(myModel->getOperand2());
60     entryFieldResult.setText(myModel->getResult());
61 }
62 }
```

In the following section, we detail each line of code:

1. Line 8 retrieves a pointer to the current model using the `getModelPointer` member function. This pointer is used to access the model's data and update the metafile accordingly.
2. Line 9 asserts the model pointer is not null. This is a safety precaution you should consider while writing your code.
3. Line 11 checks the metafile Boolean. If it is set to `TRUE`, your server is not active within a container and you must provide its cached representation. If it is `FALSE`, the server is active either in-place or out-place within the container and you must update the view entry fields with the model's data. The rest of the code consists of drawing the model's metafile.
4. Line 12 defines some `IString` objects that will be used throughout the code.
5. Line 13 acquires a graphic context from the presentation space to draw in the metafile area.
6. Line 15 sets the current fill color to white for the graphic context area.
7. Line 16 defines a rectangle that surrounds the different graphic strings displayed.
8. Lines 17 through 21 initialize the strings `str1`, `str2`, and `str3` with the `t` resources for the `Operand1`, `Operand2`, and `Result` static texts.
9. Lines 24 through 26 define three graphic text objects that display the three static texts in the rectangle defined previously.

10. Lines 28 through 42 check the length of the model's data. For each data attribute, if the length is not null, then the operand is retrieved using the pointer to the model and the accessor member functions; otherwise, a string is set to "empty" as defined in the resource file for the STR\_EMPTY resource.
11. Lines 44 through 46 define three graphic text objects that are initialized to the model's data or the empty string according to the data contents.
12. Lines 48 through 54 draw the metafile using the graphic context.
13. Lines 58 through 60 update the view according to the contents of the model's data. These lines are needed if you want your view to be displayed with the correct data when you load a previously saved document that contains a calculator server component and when you activate the component by double-clicking on it for the first time.

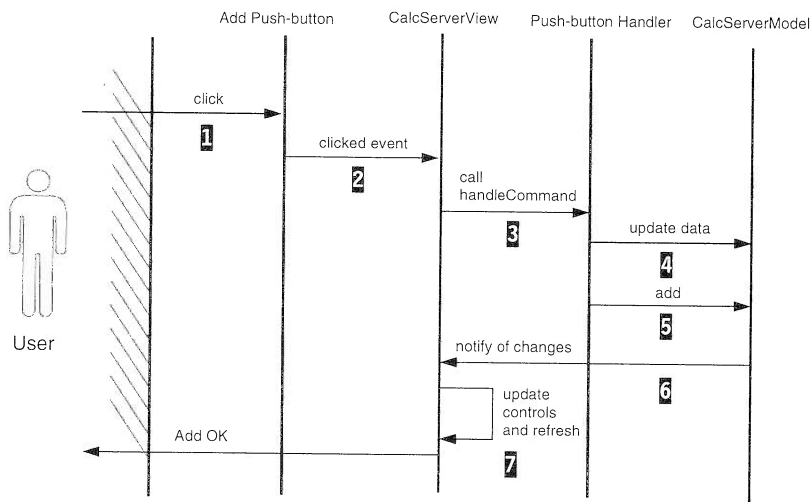
In order to manipulate `IGString`, `IGRectangle`, and `IGraphicContext` classes and to take advantage of the `assert` function, you also need to include three more header files in your `.cpp` file, as follows:

```
/*
 *COMPONENT_NAME: ClcSrvrV.cpp
 *
 *FUNCTIONS:
 *
 *PURPOSE:
 */
#include <ireslib.hpp>
#include <iframe.hpp>
#include "ClcSrvrM.hpp"
#include "ClcSrvrV.hpp"
#include <assert.h>
#include <igstring.hpp>
#include <igrect.hpp>
#include <igrafctx.hpp>
```

So far, your model and view are almost built but the infrastructure needed to let them communicate with each other at run time is still to be implemented. By now, you should know how to access the model's data from the view, but you still need a mechanism that notifies the view to repaint each time the model's data change.

## Handling Communications Between Model and View

Model and View use two different mechanisms to communicate with each other. The View communicate with the model through event handling and by using the pointer returned by the *getModelPointer* function. In return, the model needs to notify the view of any changes that its data undergo. If we consider our sample calculator server, we can draw the event flow between the different parties as shown in the event trace diagram in Figure 167.



**Figure 167.** Addition Event Trace Diagram

CalcServerView intercepts the button click event of the Add push button to update the model's data. When the user clicks the Add push button, **1**, it generates a button click event which is identified by the view using its resource identifier `ID_BUTTON`, **2**. The view then calls the *CalcServerView::handleCommand* member function to process the event, **3**. This member function sets the model's data, `operand1` and `operand2`, to the values contained in the corresponding entry fields, and then calls the *CalcServerModel::add* member function of the model to calculate the sum of its operands and update the result accordingly, **4** and **5**. Once the result data member of the model is updated, the model notifies the view to display the result, **6**. To notify the view, the model uses notification identifiers that help the view to keep track of the changes. These identifiers are defined as *INotificationId* and sent to the view as part of notification events in calling *CalcServerModel::notifyOfChange* in the appropriate model member functions. The view can track changes to the associated model's data

using the *CalcServerView::handleNotification* member function. This member function receives the notification events sent by the model and updates the view controls accordingly, **7**.

To handle the push button click event within *CalcServerView*, modify the *ClcSrvrV.cpp* implementation file as follows:

```
Boolean CalcServerView::handleCommand( ICommandEvent& cmdEvent )
{
    //////////////////////////////////////
    // ADD Event Processing here
    //////////////////////////////////////
    Boolean eventProcessed;
    CalcServerModel* myModel = getModelPointer();
    assert ( myModel != NULL );
    switch (cmdEvent.commandId()) {
        case ID_BUTTON:
            myModel->setOperand1(entryFieldOperand1.text());
            myModel->setOperand2(entryFieldOperand2.text());
            myModel->add();
            eventProcessed = true;
            break;
        default:
            eventProcessed = false;
    }
    return eventProcessed;
}
```

Notice that the resource identifier, *ID\_BUTTON*, is used to identify the event received by the view. Then using the pointer on the model returned by *getModelPointer*, *operand1* and *operand2* are updated using their set accessor member functions. At last, the *add* model member function is called to add the two operands and update the result.

You add model notifications to *CalcServerModel* in three steps:

1. For each data member, declare the static notification identifier in the public section of the *CalcServermodel* declaration and add the *inotifev.hpp* include file to its headers:

```
#ifndef _CLCSRVM_
#define _CLCSRVM_

/*
 *COMPONENT_NAME: ClcSrvrM.hpp
 *
 *FUNCTIONS: mainly setter and getter functions for the attributes
 *
 *PURPOSE:
 */

#include <inotifev.hpp>
#include <istring.hpp>
```

```
#include <imodel.hpp>

//-----
// CalcServerModel
//-----

class CalcServerModel : public IModel
{
public:
    TypeExtensionDeclarationsMacro(CalcServerModel)

    static const INotificationId operand1Change;
    static const INotificationId operand2Change;
    static const INotificationId resultChange;

    CalcServerModel();
    virtual ~CalcServerModel();
};
```

2. Define the notification identifiers in the .cpp file after the macro declaration:

```
TypeExtensionMacro(CalcServerModel)

const INotificationId CalcServerModel::operand1Change =
    "Operand1 Changed";
const INotificationId CalcServerModel::operand2Change =
    "Operand2 Changed";
const INotificationId CalcServerModel::resultChange =
    "Result Changed";
```

3. Add a **notifyOfChange** call wherever the model's data change, by using the proper notification identifier:

```
Boolean CalcServerModel::setOperand1(const IString& str)
{
    operand1 = str;
    notifyOfChange(INotificationEvent(operand1Change, notifier()));
    return true;
}

Boolean CalcServerModel::setOperand2(const IString& str)
{
    operand2 = str;
    notifyOfChange(INotificationEvent(operand2Change, notifier()));
    return true;
}

Boolean CalcServerModel::setResult(const IString& str)
{
    result = str;
    notifyOfChange(INotificationEvent(resultChange, notifier()));
    return true;
}
```

Notice that the *add* member function modifies the result data member using the set accessor *setResult* which in turn sends the notification to the view in order to update the contents of the result entry field.

Now that your model and view are completed, you need to glue them together using a *stationery* object. We detail this last step in the next section.

## Instantiating the Stationery and Creating the Main Function

The *stationery* provides your server with a frame window, a toolbar and all services needed to run your application, including instantiating the application, starting the message pump and so forth. To create your component *stationery* instance, you do not need to write any code, since Project Smarts did it for you in inserting in `clcsrvrm.cpp` the following line:

```
IComponentStationeryFor<CalcServerModel, CalcServerView> calcServerStationery
```

You can customize the component *stationery* by providing new menu items to the frame window or a new toolbar button through overriding the *IComponentStationery*. For more information, consult the Visual-Age for C++ Open Class Library User's Guide.

The last step is already done by Project Smarts which generates a *main* function for your server in your model source. This function only executes the *run* member function of the *CalcServerStationery* object you just created:

```
int main( int argc, char* argv[] )
{
    return calcServerStationery.run( argc, argv );
}
```

## Compiling and Linking Your Application

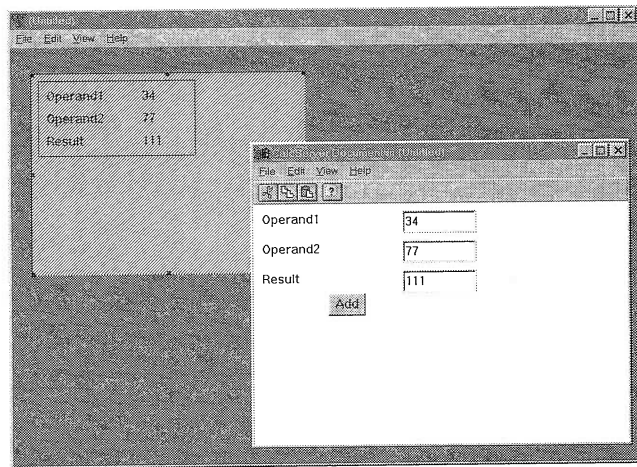
In your WorkFrame project menu bar, select **Project** → **Build normal**, or use the accelerator key **Ctrl+Shift+B** to generate the make file, compile, and link-edit your application.

The final step is to run your server component as a stand-alone, so that its CSLID is registered in the registry file of your operating system, thereby making your application available as an OLE object.

## Using Your OLE Server

To use your new component, you need an OLE container application. You can use the container sample provided in the VisualAge for C++ samples for the IBM Open Class Libraries (this container is located in X:\IBMCPPW\SAMPLES\IOC\ACDF3 and the file is ACDF3.EXE) as follows:

1. Start the ACDF3 container and select **Insert** → **Insert Object...** from the toolbar menu
2. Select **CalcServer Document** in the Object Type list box to insert your calculator component (if your component is not listed in the list box, it is because it is not yet registered in the registry and you must run it once as a stand-alone), then click the **OK** push button. The component is embedded in the container at activated in place.
3. Enter new operand values and click the **Add** push button. The result is displayed in the corresponding entry field.
4. Now click off the component. The component is deactivated and the metafile is displayed instead.
5. Select the component by clicking on it and click with the right mouse button to display the component's pop-up menu. Select **CalcServer** → **Open** to activate the component out-place. Notice that the metafile is grayed out (see Figure 168).
6. Modify the operand values again and click the Add push button. The result is displayed again in the result entry field and the metafile is updated simultaneously. Close the CalcServer component.
7. Select **File** → **Save As...** from the menu bar of the container. Enter a filename in the dialog box and click the **OK** push button to save your document.
8. Click the first button of the container's toolbar to create a new document, then click the second button from the toolbar to retrieve the document you just saved. Select your document in the dialog box and validate your choice with the **OK** push button. The document is loaded into the container and the model's data are displayed in the view.
9. Close the container.



**Figure 168.** Out-Place Activation of the Calculator Server



Congratulations! You have built your first OLE server. As you can see, the framework provides you with most of the functionality needed for free and the amount of code necessary to customize the behavior of your server is not excessive.

If you pay attention to the notification framework that supports the model and the view, and if you refer to Chapter 10, “More about Visual Builder...”, you will notice that the notification framework of the Visual Builder is based on the same notification principles. In fact, the *notifyOfChange* member function which is called in your model accessor member functions calls in turn the *notifyObservers* member function of any *INotifier*. It is then possible to take advantage of the construction from parts of the Visual Builder to build up the view and the model of an OLE component. This is an undocumented feature that we detail in the next section.

## Developing OLE Components with the Visual Builder

The Visual Builder is the main tool that helps you to develop applications using the building-from-parts paradigm. As you now know, the Visual Builder leads you to a clear separation between the logic of your application and its representation. The logic is built from nonvisual or class interface parts and the representation is built using visual parts. The parts are connected with each other to implement the notification plumbing necessary and to trigger, from the GUI interface, the services of the nonvisual parts.



From this perspective, you can see that a view built from visual parts could be reused as the view of an OLE component. Likewise, the logic of your application, implemented from the nonvisual parts, could be reused as a model in an OLE component.

To illustrate how you can build an OLE component using the Visual Builder, you build an OLE SimulMortgage server, from the ASimulMortgageView and AMortgageCalculator parts that you built in Chapter 7, 8 and 9.

## Creating Your Project with WorkFrame

To create a project skeleton for your SimulMortgage server, use WorkFrame as follows:

1. In the IBM VisualAge for C++ for Windows container, double-click the **WorkFrame IDE** icon
2. Push the **Create new project** push button. The Project Smarts notebook is displayed on the screen.
3. Scroll down the Project types list, select the **Compound Document Framework** type, and click the **Next** push-button.
4. In the Location page, fill out the entry fields as follows:

<b>Project Title</b>	SimulMortgage Server
<b>Source File Directory</b>	D:\IBMCPW\WORKING\OLESIMUL
<b>Where to Create the Project Files</b>	D:\IBMCPW\WORKING
<b>Project File Name</b>	OLESIMUL
5. Click the **Next** push-button
6. In the Component page, select **Server** as the Component type and fill in the Component name entry field with **SimulMortgage**. Then click the **Next** push button.
7. In the Application Name page, **SimulMortgage** is selected as the Application Short Name. Likewise, **SimulMortgage Document** is automatically selected as the Application Long Name. Click the **Next** push button
8. In the Model Class page, **SimulMortgageModel** is automatically selected as your model class name. Click the **Next** push button.
9. In the View Class page, **SimulMortgageView** is selected as your view class name. Click the **Next** push button.
10. In the File Extension page, the **sim** extension is automatically proposed for all files that are saved and loaded by the SimulMortgage component. Click the **Next** push button.

11. In the Prologue page fill-in the different entry fields to your liking and click the **Done** push button.

Your project is created and should now contain the the files listed in Table 75.

<b>Table 75.</b> File List for the SimulMortgage OLE Component Project	
<b>Source file</b>	<b>File</b>
MrtggSr3.hpp	MortgageServerView header file
MrtggSr3.cpp	MortgageServerView implementation file
MrtggSr2.hpp	MortgageServerModel header file
MrtggSr2.cpp	MortgageServerModel implementation file
MrtggSrv.reg	Component registration file
MrtggSrv.rc	Component resource file
Readme.txt	Component readme file

To build your OLE component and use the parts generated by the Visual Builder, execute the following steps:

1. Modify the visual part code generation to externalize its resources. In effect, you need to access the string resources of your application to build the metafile of your component. You also need to promote the controls the view must interact with.
2. Modify the class interface part to access your model's data. In effect, the class interface part must be able to update the model's data since it represents an extension of the model.
3. Modify the component model code generated by Project Smarts to add the model's data definitions that need to be persistent, their notification identifiers, and the corresponding accessor member functions.
4. Modify the component view code generated by Project Smarts to use the visual part as the view client area and handle the model notifications when the model's data change.

In the following sections we detail each of these steps.

## Modifying the Visual Part

Two modifications are necessary to attach your visual part as the client area of your component's view:

- ❑ The resources of your visual part must be externalized to allow the view to access it when building the component's metafile.
- ❑ The four entry fields must be promoted to allow the view to access and modify their contents when drawing the component.

To externalize your visual part resources,

1. Start the Visual Builder from your project and load the VBB file which contains ASimulMortgageView.
2. Edit the visual part by double-clicking ASimulMortgageview in the main browser window. The part is loaded into the free-form surface.
3. Switch to the Class Editor and check the mark **Starting resource id** to force Visual Builder to generate the resource identifiers of your controls starting from **10000**.

To promote the entry fields,

1. Switch to the Composition Editor.
2. Select the **Promote Part Feature...** menu item from the pop-up menu of EntryFieldIncome.
3. Select the **this** attribute in attribute list box.
4. Click the **Promote** push button to promote the attribute.
5. Repeat Steps 2 through 4 for the remaining entry fields except EntryFieldEstimation which has been already promoted in Chapter 7.

Save the part and generate its code by selecting the **File → Save and Generate Part source** option in the Visual Builder menu bar. The source code for the part is generated in your project directory along with the resource file `vrssimv.rci`.

Before modifying the code of AMortgageCalculator, you need to copy manually its implementation files, `vrssimc.hpp` and `vrssimc.cpp`, to your project. Because AMortgageCalculator is a class interface part, you cannot have access to the **Generate Part Source** option to generate those files automatically in your project.

At last, you must copy the files `kbdhdr.hpp` and `kbdhdr.lib` in your project in order to compile and link-edit the code related to ASimulMortgageView (see “Event Handler” on page 237 for more information regarding the keyboard event handlers attached to ASimulMortgageView).

Your project should now contain the files listed in Table 76.

**Table 76.** File List for the SimulMortgage OLE Component Project

Source file	File
MrtggSr3.hpp	MortgageServerView header file
MrtggSr3.cpp	MortgageServerView implementation file
MrtggSr2.hpp	MortgageServerModel header file
MrtggSr2.cpp	MortgageServerModel implementation file
MrtggSrv.reg	Component registration file
MrtggSrv.rc	Component resource file
Readme.txt	Component readme file
vrssimv.h	ASimulMortgageView resource header file
vrssimv.rci	ASimulMortgageView resource file
vrssimv.hpp	ASimulMortgageView header file
vrssimv.cpp	ASimulMortgageView implementation file
vrssimc.hpp	AMortgageCalculator header file
vrssimc.cpp	AMortgageCalculator implementation file
kbdhdr.hpp	Keyboard handlers header file
kbdhdr.lib	Keyboard handlers library

## Modifying the Class Interface Part

In order to use the class interface part as an extension of your model component, you must provide the code to access the model's data and update it when necessary.

1. Edit the `vrssimc.hpp` file and add the following include file statements:
  - `#include "MrtggSr2.hpp"`
  - `#include <istatnry.hpp>`
  - `#include <iguibndl.hpp>`
  - `#include <icompenh.hpp>`
2. Edit the `vrssimc.cpp` file and modify the *AMortgageCalculator::Estimate* member function by adding the information shown in bold:

```
double AMortgageCalculator::Estimate()
{
    double result = inome*30./100*exp(rate/100*years);
```

```

        ((MortgageServerModel*)IComponentStationery::stationery()).
            bundle().
            component().
            model()->setEstimation(result);
    setEstimation(result);
    return result;
} // ADD your public member functions here

```

The *IComponent::stationery()* static call returns the stationery instance of your component. From this instance, you can access the GUI bundle object and thereby access the associated component. Finally, you can access the model of your component and call the accessor member function to set the estimation data member.

## Modifying the Server Model Code

To reuse the class interface part *AMortgageCalculator* in your model code, you must add to your model its data attributes and their accessor member functions. This is necessary to allow the data to be persistent and saved to a structured storage file. Then, for each attribute you must attach a notification identifier that will be used to notify the view accordingly.

Open the *MrtggSr2.hpp* file and add the following line of codes (the lines to add are shown in bold):

```

#ifndef _MRTGGSR2_
#define _MRTGGSR2_
/*
 * COMPONENT_NAME: MrtggSr2.hpp
 *
 * FUNCTIONS:
 *
 * PURPOSE:
 */
#include <imodel.hpp>
#include <istring.hpp>
#include <inotifev.hpp>
//-----
// MortgageServerModel
//-----
class MortgageServerModel :
public IModel
{
public:
    TypeExtensionDeclarationsMacro(MortgageServerModel)
    static const INotificationId incomeChange;
    static const INotificationId yearsChange;
    static const INotificationId rateChange;
    static const INotificationId estimationChange;
public:
    MortgageServerModel();

```

```

virtual ~MortgageServerModel();
virtual IBaseStream& operator>>=( IBaseStream& toWhere ) const;
virtual IBaseStream& operator<<=( IBaseStream& fromWhere );
// ADD your public member functions here
virtual double      getIncome() const;
virtual IBoolean    setIncome(const int&);
virtual double      getYears() const;
virtual IBoolean    setYears(const int&);
virtual double      getRate() const;
virtual IBoolean    setRate(const double&);
virtual double      getEstimation() const;
virtual IBoolean    setEstimation(const double&);
protected:
                                MortgageServerModel( const MortgageServerModel& copy );
private:
    // Hide assignment operator
    MortgageServerModel& operator=( const MortgageServerModel& copy );
    enum { kOriginalVersion };
    // ADD your data members here
    double income;
    double years;
    double rate;
    double estimation;
};
#endif // _MRTGGSr2_

```

Two header files are included in the code in order to manipulate strings and take advantage of the notification framework. The accessor member functions for the income, the years, the rate, and the estimation attributes are then declared. The data members of `AMortgageCalculator` are added at the bottom of the file.

You must now modify the implementation file, `MrtggSr2.cpp` to reflect the changes made in the `.hpp` file by adding the code shown in bold type:

```

/*
 * COMPONENT_NAME: MrtggSr2.cpp
 *
 * FUNCTIONS:
 *
 * PURPOSE:
 *
 */
#include "MrtggSr2.hpp"
#include "MrtggSr3.hpp"
#include <ibasstrm.hpp>
#include <istatnry.hpp>
#include <iexcept.hpp>
#include <iconst.h>

```

```

//-----
// MortgageServerModel Methods
//-----
TypeExtensionMacro(MortgageServerModel)
const INotificationId MortgageServerModel::nIncomeChange    = "Income Changed";
const INotificationId MortgageServerModel::nYearsChange     = "Years Changed";
const INotificationId MortgageServerModel::nRateChange      = "Rate Changed";
const INotificationId MortgageServerModel::nEstimationChange = "Estimation
Changed";
MortgageServerModel::MortgageServerModel()
: IModel()
{
}
MortgageServerModel::MortgageServerModel( const MortgageServerModel &copy )
: IModel( copy ) /* force assert */
{
}
MortgageServerModel::~~MortgageServerModel()
{
}
short MortgageServerModel::getIncome() const
// Implement method for reading member data
{
    IFUNCTRACE_DEVELOP();
    return income;
}
Boolean MortgageServerModel::setIncome( const double& aIncome )
// Implement method for setting member data
{
    IFUNCTRACE_DEVELOP();
    income = aIncome;
    notifyOfChange(INotificationEvent(nIncomeChange, notifier()));
    return true;
}
double MortgageServerModel::getYears() const
// Implement method for reading member data
{
    IFUNCTRACE_DEVELOP();
    return years;
}
Boolean MortgageServerModel::setYears( const double& aYears )
// Implement method for setting member data
{
    IFUNCTRACE_DEVELOP();
    years = aYears;
    notifyOfChange(INotificationEvent(nYearsChange, notifier()));
    return true;
}
double MortgageServerModel::getRate() const
// Implement method for reading member data
{
    IFUNCTRACE_DEVELOP();
    return rate;
}
Boolean MortgageServerModel::setRate( const double& aRate )
// Implement method for setting member data
{
    IFUNCTRACE_DEVELOP();
    rate = aRate;
    notifyOfChange(INotificationEvent(nRateChange, notifier()));
    return true;
}

```

```

}
double MortgageServerModel::getEstimation() const
// Implement method for reading member data
{
    IFUNTRACE_DEVELOP();
    return estimation;
}
Boolean MortgageServerModel::setEstimation( const double& aEstimation )
// Implement method for setting member data
{
    IFUNTRACE_DEVELOP();
    estimation = aEstimation;
    notifyOfChange(INotificationEvent(nEstimationChange, notifier()));
    return true;
}
IBaseStream& MortgageServerModel::operator>>= ( IBaseStream& toWhere ) const
{
    writeVersion( toWhere );
    IModel::operator>>= ( toWhere );
    //////////////////////////////////////
    // ADD stream out of data members here
    //////////////////////////////////////
    income    >>= toWhere;
    years     >>= toWhere;
    rate      >>= toWhere;
    estimation >>= toWhere;
    return toWhere;
}
IBaseStream& MortgageServerModel::operator<<= ( IBaseStream& fromWhere )
{
    switch ( readVersion( fromWhere ) ) {
    case kOriginalVersion:
        IModel::operator<<= ( fromWhere );
        //////////////////////////////////////
        // ADD stream in of data members here
        //////////////////////////////////////
        income    <<= fromWhere;
        years     <<= fromWhere;
        rate      <<= fromWhere;
        estimation <<= fromWhere;
        break;
    default:
        ITHROWLIBRARYERROR(IC_STREAM_VERSION_UNSUPPORTED,
                           IBaseErrorInfo::invalidRequest,
                           IException::recoverable);
    }
    return fromWhere;
}
//-----
// Template Instantiation
//-----
IComponentStationeryFor<MortgageServerModel,MortgageServerView>
mortgageServerStationery;
int main( int argc, char* argv[] )
{
    return mortgageServerStationery.run( argc, argv );
}

```



As you may notice, the implementation code for the `MortgageServerModel` class is similar to the implementation code of the calculator model. The few lines added at the beginning define the notification identifiers of the model's data. Then the accessor member functions are defined (notice the call to `notifyOfChange` in the `setIncome`, `setYears`, `setRate`, and `setEstimation` member functions). Finally, the streaming operators are modified in order to save and load the model's data.

## Modifying the Server View Code

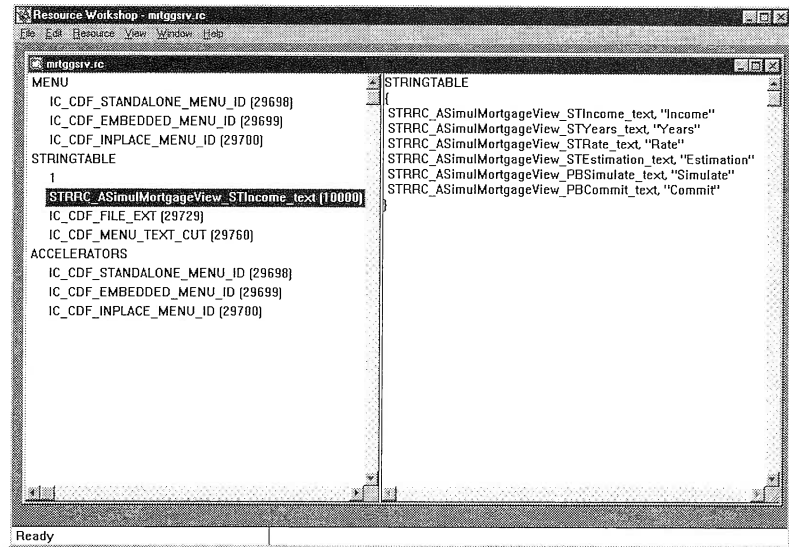
The server view of the `MortgageServerView` part is entirely provided by the visual part, `ASimulMortgageView`, that you design using the Visual builder. To use this part as the view of your component you must associate `ASimulMortgageView` with `MortgageServerView` using an instance variable and modifying the resource file of your component in order to include the resources of the visual part that you externalize in "Modifying the Visual Part" on page 493.

To include the resources of `ASimulMortgageView` in the resource file of your component, edit the file `MrtggSrv.rc` by selecting `Edit` from its pop-up menu, and add the following line at the bottom:

```
#include "vrssimv.rci"
```

This line includes the resource script file generated by Visual Builder for the visual part `ASimulMortgageView`. When the resource compiler processes the file `MrtggSrv.rc`, it appends the resources of `ASimulMortgageView` (mostly string tables) to `MrtggSrv.rc` and compiles the file to produce the binary file `MrtggSrv.res`.

You can check that the resources are added to the file `MrtggSrv.rc` by double-clicking to invoke the resource editor (Figure 169). Notice that the string table contains the resources associated to `ASimulMortgageView`. The resource identifiers are prefixed by `STRRC_ASimulMortgageView`. Refer to these resource identifiers when you code your component view.



**Figure 169.** Resource Workshop Window

To include ASImulMortgageView as an instance variable of MortgageServerView, modify the MrtggSr3.hpp file by adding the code shown in bold:

```
#ifndef _MRTGGSR3_
#define _MRTGGSR3_
/*
 * COMPONENT_NAME: MrtggSr3.hpp
 *
 * FUNCTIONS:
 *
 * PURPOSE:
 */
#include <iview.hpp>
#include <icmdhdr.hpp>
#include "vrssimv.hpp"

class MortgageServerModel; // forward declaration for getModelPointer
//-----
// MortgageServerView
//-----
class MortgageServerView : public IView
{
public:
    virtual ~MortgageServerView();
    MortgageServerView( IGUIBundle& bundle );
    virtual void initialize();
    virtual void finalize();
}
```

```

        virtual Boolean      handleCommand( ICommandEvent& cmdEvent );
        ///////////////////////////////////////////////////
        //  ADD your public member function here
        ///////////////////////////////////////////////////
        virtual MortgageServerModel*  getModelPointer() const;
        virtual void handleNotification( const INotificationEvent& event);

protected:
        virtual void          drawContents( IPresSpaceHandle& ps,
                                           const IRectangle&,
                                           Boolean );

private:
        // Hide assignment operator
        MortgageServerView& operator=( const MortgageServerView& copy );
        // Command Connection for this View
        ICommandConnectionTo<MortgageServerView> fCommandHandler;
        ///////////////////////////////////////////////////
        //  ADD handlers, controls, and other data here
        ///////////////////////////////////////////////////
        ASimulMortgageView mySimulMortgageView;
    };
#endif // _MRTGGSr3_

```

At the top of the file, you include the header file `vrssimv.hpp` which contains the definition of the `ASimulMortgageView` visual part. Then the `MortgageServerModel` class is forward-declared to resolve the return type of the `getModelPointer` member function. You declare this function in your code to access your model's data from the view. The *handleNotification* member function is also declared to refresh the view when the model notifies it of data changes. At last, the most important line is the declaration of `mySimulMortgageView` instance variable which holds the client area of the server view. This instance is constructed in the `MortgageServerView` constructor and assigned to your model view as a client area.

You must now modify the view implementation code to reflect the changes in the header file by adding the code shown in bold:

```

001 // Hide assignment operator
002 /*
003  *COMPONENT_NAME: MrtggSr3.cpp
004  *
005  *FUNCTIONS:
006  *
007  *PURPOSE:
008  *
009  */
010 #include "MrtggSr3.hpp"
011 #include "MrtggSr2.hpp"
012 #include "vrssimv.h"
013 #include <assert.h>
014 #include <notifev.hpp>
015 #include <iframe.hpp>

```

```

016 #include <igstring.hpp>
017 #include <igrect.hpp>
018 #include <igrafctx.hpp>
019
020 //-----
021 // MortgageServerView Methods
022 //-----
023 MortgageServerView::~MortgageServerView()
024 {
025 }
026
027 MortgageServerView::MortgageServerView( IGUIBundle& bundle )
028 : IView( bundle ),
029 mySimulMortgageView(WND_ASimulMortgageView_MultiCellCanvas, this, this),
030 fCommandHandler( *this, handleCommand )
031 {
032     mySimulMortgageView.initializePart();
033 }
034
035 void MortgageServerView::initialize()
036 {
037     IView::initialize();
038     //////////////////////////////////////
039     // ADD initialization code here
040     //////////////////////////////////////
041     MortgageServerModel* myModel = getModelPointer();
042     assert(myModel != NULL);
043     setViewClient( &mySimulMortgageView );
044     mySimulMortgageView.getEntryFieldRate()->
045         setText( IString(myModel->getRate()));
046     mySimulMortgageView.getEntryFieldIncome()->
047         setText( IString(myModel->getIncome()));
048     mySimulMortgageView.getEntryFieldYears()->
049         setText( IString(myModel->getYears()));
050     mySimulMortgageView.getEntryFieldEstimation()->
051         setText( IString(myModel->getEstimation()));
052     fCommandHandler.handleEventsFor( this );
053 }
054
055 void MortgageServerView::finalize()
056 {
057     fCommandHandler.stopHandlingEventsFor( this );
058     //////////////////////////////////////
059     // ADD finalization code here
060     //////////////////////////////////////
061     IView::finalize();
062 }
063
064 MortgageServerModel* MortgageServerView::getModelPointer() const
065 // Get the pointer to the model
066 {
067     MortgageServerModel* theModel = NULL;
068     ::dynamicCastTo( theModel,model() ); // Downcasts the model pointer
069     return theModel;
070 }

```

```

071
072 Boolean MortgageServerView::handleCommand( ICommandEvent& cmdEvent )
073 {
074     //////////////////////////////////////
075     // ADD Event Processing here
076     //////////////////////////////////////
077     MortgageServerModel* myModel = getModelPointer();
078     assert(myModel != NULL);
079     Boolean result = false;
080     switch(cmdEvent.commandId()) // Catch the button pressed and menu commands
081     {
082     case WND_ASimulMortgageView_PBSimulate :
083     {
084         myModel->setRate(mySimulMortgageView.getEntryFieldRate()->
085             text().asDouble());
086         myModel->setIncome(mySimulMortgageView.getEntryFieldIncome()->
087             text().asDouble());
088         myModel->setYears(mySimulMortgageView.getEntryFieldYears()->
089             text().asDouble());
090         result = true;
091         break;
092     }
093     }
094     return result;
095 }
096
097 void MortgageServerView::drawContents( IPresSpaceHandle& ps,
098     const IRectangle&,
099     Boolean metafile)
100 {
101     //////////////////////////////////////
102     // ADD Drawing of your view here
103     //////////////////////////////////////
104     MortgageServerModel* myModel = getModelPointer();
105     assert(myModel != NULL);
106     if (metafile)
107     { IString str1, str2, str3, str4, ef1, ef2, ef3, ef4;
108         IGraphicContext ctxt (ps );
109         ctxt.setFillColor(IColor::white);
110         IRectangle myRect(IRectangle(IPoint(10,10),IPoint(220,140)));
111         str1 = IApplication::current().
112             userResourceLibrary().
113             loadString(STRRC_ASimulMortgageView_STIncome_text);
114         str2 = IApplication::current().
115             userResourceLibrary().
116             loadString(STRRC_ASimulMortgageView_STYears_text);
117         str3 = IApplication::current().
118             userResourceLibrary().
119             loadString(STRRC_ASimulMortgageView_STRate_text);
120         str4 = IApplication::current().
121             userResourceLibrary().
122             loadString(STRRC_ASimulMortgageView_STEstimation_text);
123         IGString static1(str1,IPoint(20,20));
124         IGString static2(str2,IPoint(20,50));
125         IGString static3(str3,IPoint(20,80));

```

```

126     IGString static4(str4,IPoint(20,110));
127     ef1 = IString(myModel->getIncome());
128     ef2 = IString(myModel->getYears());
129     ef3 = IString(myModel->getRate());
130     ef4 = IString(myModel->getEstimation()).subString(1,7);
131     IGString gef1(ef1,IPoint(150,20));
132     IGString gef2(ef2,IPoint(150,50));
133     IGString gef3(ef3,IPoint(150,80));
134     IGString gef4(ef4,IPoint(150,110));
135     myRect.drawOn(ctxt);
136     static1.drawOn( ctxt );
137     static2.drawOn( ctxt );
138     static3.drawOn( ctxt );
139     static4.drawOn( ctxt );
140     gef1.drawOn( ctxt );
141     gef2.drawOn( ctxt );
142     gef3.drawOn( ctxt );
143     gef4.drawOn( ctxt );
144 }
145 else
146 {
147     mySimulMortgageView.getEntryFieldRate()->setText(IString(myMode->
148                                     getRate()));
149     mySimulMortgageView.getEntryFieldIncome()->setText(IString(myModel->
150                                     getIncome()));
151     mySimulMortgageView.getEntryFieldYears()->setText(IString(myModel->
152                                     getYears()));
153     mySimulMortgageView.getEntryFieldEstimation()->setText(IString(myModel->
154                                     getEstimation()));
155 }
156 }
157
158 void MortgageServerView::handleNotification(const INotificationEvent&event)
159 // Handle the notification events
160 {
161     MortgageServerModel* myModel = getModelPointer();
162     assert(myModel != NULL);
163     if (event.notificationId() == myModel->nIncomeChange )
164     {
165         mySimulMortgageView.getEntryFieldIncome()->setText(IString(myModel->
166                                     getIncome()));
167     }
168     else if (event.notificationId() == myModel->nYearsChange )
169     {
170         mySimulMortgageView.getEntryFieldYears()->setText(IString(myModel->
171                                     getYears()));
172     }
173     else if (event.notificationId() == myModel->nRateChange )
174     {
175         mySimulMortgageView.getEntryFieldRate()->setText(IString(myModel->
176                                     getRate()));
177     }
178     else if (event.notificationId() == myModel->nEstimationChange )
179     {
180         mySimulMortgageView.getEntryFieldEstimation()->setText(IString(myModel->

```

```
180                                     getEstimation()));  
181     }  
182     refresh(); // will refresh the screen  
183 }
```

The lines of code (in bold) that you must add to the original file serve several purposes:

1. Lines 11 through 18 include header files that are necessary either to access the code of `ASimulMortgageview`, to draw the component metafile or to handle the notification sent by the component model.
2. Line 29 constructs the `mySimulMortgageView` instance variable. Notice that you transmit the proper resource identifier, `WND_ASimulMortgageView_MultiCellCanvas`, as the first parameter of the constructor.
3. Line 32 initializes the visual part in order to handle the notification for each entry field, connect the different parts, and notify the potential observers, using the `readyEventId`, of part readiness. Without this line of code, clicking on the **Add** push button would not have any effect since the notification infrastructure would not be set up properly (for more information See Chapter 10, “More about Visual Builder...”).
4. Lines 41 through 51 initialize the server view according to the model’s data.
5. Lines 64 through 70 implement the *getModelPointer* function, which retrieves a pointer to the server model.
6. Lines 77 through 94 implement the *handleCommand* member function. You do not need to call the *AMortgageCalculator::Estimate()* member function. In effect, the click event notification of the push button is handled automatically by the connections drawn between `ASimulMortgageView` and `AMortgageCalculator`. In this case, only one connection starts the execution of the *AMortgageCalculator::Estimate()* member function, but this same sequence can be used to program a more complex algorithm for your server model using the building-from-parts paradigm of the Visual Builder.
7. Lines 104 through 154 override the *drawContents* member function to provide the component with a metafile and refresh the view according to the model’s data.
8. Lines 157 through 183 handle the notification sent by the model when the data change.

As you can see the code is very similar to the code you develop in the Calculator example. Meanwhile, you benefit from the Visual Builder from two perspectives:

- ❑ You do not need to code your server view; the Visual Builder automatically generates the code for you.
- ❑ You can program the logic of your server model visually using the Visual Builder and let the CDF handle the persistent functionality of your model's data.

You can now build your server by selecting **Project** → **Build normal**, (or use the accelerator key **Ctrl+Shift+B**) from your WorkFrame project menu bar.



You also need to run your server component as a stand-alone so that its CSLID is registered in the registry file of your operating system, thereby making your application available as an OLE object.





# A

## Installing the Application

### Warning



Before installing the application make sure that the following products are installed on your machine:

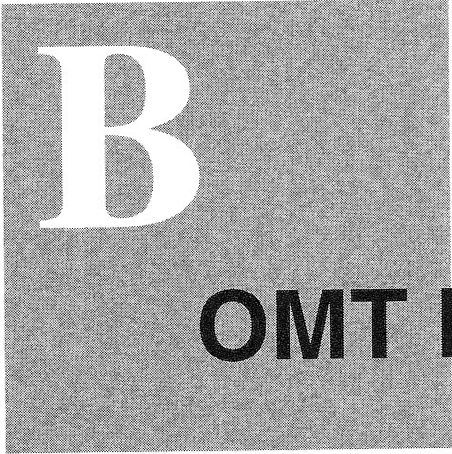
- ☐ DATABASE 2™ for Windows NT or Windows 95 (depending on your machine). If DB2 for Windows NT or Windows 95 is already installed on your system, make sure that the DB2 Software Developer's Kit is also installed.)
- ☐ Windows NT 3.51 or Windows 95.
- ☐ VisualAge™ for C++ V3.5 for Windows (all components)

All of the files that you need to run the sample application are stored in the CD-ROM that accompanies this book. If you do not have the required products, the CD-ROM provides you with a trial version of DB2™ for Windows Version 2.1.1 (DB2 Software Developer's Kit included) and VisualAge for C++ for Windows Version 3.5.

---

To make it easy for you to install the Visual Realty application, we have included three installation programs on the CD-ROM itself for each component (DB2, VisualAge for C++, and the sample application). Each installation program is located in a specific directory on the CD-ROM.

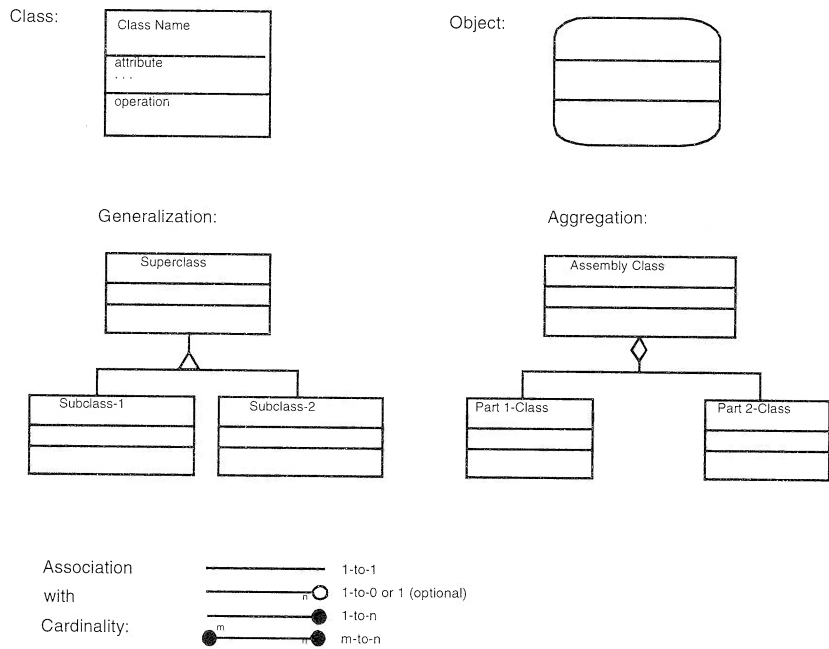
Consult the READ.ME file located on your CD-ROM for the latest information regarding the sample application and the installation procedure.



# B

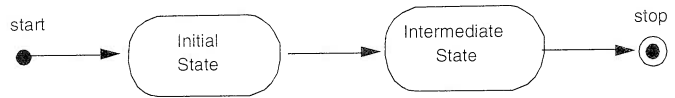
# OMT Notation

In this appendix we show the schema representation of the object model (see Figure 170) and the state diagram (see Figure 171 ) for the Object Modeling Technique method.



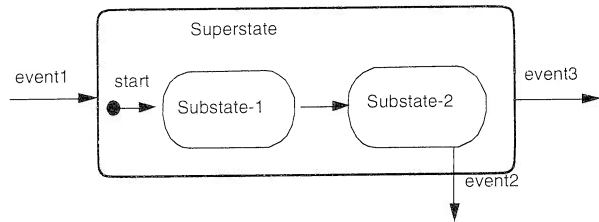
Source: Object-Oriented Modeling and Design - James Rumbaugh et al.

**Figure 170.** OMT Notation: Object Model



Arc label of the form:

Event(attribute) [Condition] / Action



Source: Object-Oriented Modeling and Design - James Rumbaugh et al.

**Figure 171.** OMT Notation: State Diagram

---

# C

## Database Definition

This appendix lists the data definitions that define the layout of the relational tables used in the Visual Realty application. The left column of Table 77 shows the SQL statements that build the DB2 database. The right column shows the corresponding SQL statements to create the dbf files. They base on the data types of the INTERSOLV ODBC dBase driver. Differences between both definitions are highlighted. Dbf files do not support the concept of views within the database.



**Table 77.** Database Layout for the REAL Database

DB2 Table/View	Dbf File
<b>Buyer</b>	<b>Buyer.dbf</b>
create table <b>userid.buyer</b> ( <b>buyer_id</b> char(11) <b>not null</b> , first_name char(20), last_name char(20), marrital_status char(1), gender char(1), income decimal(7,0), work_phone char(12), home_phone char(12), <b>primary key (buyer_id) );</b>	create table buyer ( <b>buyer_id</b> char(11), first_name char(20), last_name char(20), <b>marrital</b> char(1), gender char(1), income <b>numeric</b> (7,0), work_phone char(12), home_phone char(12));
<b>Buyer_address</b>	<b>BuyAdd.dbf</b>
create table <b>userid.buyer_address</b> ( <b>address_id</b> char(11) <b>not null</b> , street char(40), area char(40) <b>not null</b> , city char(40), state char(2), zip_code char(5), <b>primary key (address_id));</b>	create table <b>buyadd</b> ( <b>address_id</b> char(11), street char(40), area char(40), city char(40), state char(2), zip_code char(5));
<b>Buyer_log</b>	<b>BuyLog.dbf</b>
create table <b>userid.buyer_log</b> ( <b>buyer_id</b> char(11) <b>not null</b> , creation_timestamp timestamp, last_update timestamp, <b>primary key (buyer_id) );</b>	create table <b>buylog</b> ( <b>buyer_id</b> char(11), <b>creation</b> char(25), <b>update</b> char(25) );
<b>Marketing_info</b>	<b>MarkInfo.dbf</b>
create table <b>userid.marketing_info</b> ( <b>property_id</b> char(5) <b>not null</b> , price decimal(8,0) <b>not null</b> , days_on_market smallint, commission_rate decimal(4,2), down_payment_rate decimal(4,2), <b>primary key (property_id) );</b>	create table <b>markinfo</b> ( <b>prop_id</b> char(5), price <b>numeric</b> (8,0), <b>days</b> <b>numeric</b> (3,0), <b>commission</b> <b>numeric</b> (4,2), <b>down_pay</b> <b>numeric</b> (4,2));
<b>Multidoc</b>	<b>Multidoc.dbf</b>
create table <b>userid.multidoc</b> ( <b>multidoc_id</b> char(5) <b>not null</b> , filename varchar(254) <b>not null</b> , type char(20) <b>not null</b> , <b>primary key (multidoc_id) );</b>	create table multidoc ( <b>doc_id</b> char(5), filename char(254), type char(20));

**Table 77.** Database Layout for the REAL Database

DB2 Table/View	Dbf File
Preference	Prefer.dbf
create table userid.preference (buyer_id char(11) not null , max_price decimal(8,0), min_price decimal(8,0), max_size decimal(5,0), min_size decimal(5,0), bedrooms smallint, bathrooms smallint, stories smallint, heating char(30), cooling char(30), primary key (buyer_id) );	create table prefer (buyer_id char(11), max_price numeric (8,0), min_price numeric (8,0), max_size numeric (5,0), min_size numeric (5,0), bedrooms numeric(2,0) , bathrooms numeric(2,0) , stories numeric(2,0) , heating char(30), cooling char(30));
Property	Property.dbf
create table userid.property (property_id char(5) not null , size decimal(5,0) not null , bedrooms smallint not null , bathrooms smallint not null , stories smallint not null , heating char(30) not null , cooling char(30) not null , description varchar(512), primary key (property_id) );	create table property (prop_id char(5), size numeric (5,0), bedrooms numeric(2,0) , bathrooms numeric(2,0) , stories numeric(2,0) , heating char(30), cooling char(30), descript memo (512));
Property_address	PropAdd.dbf
create table userid.property_address (address_id char(5) not null , street char(40), area char(40) not null , city char(40), state char(2), zip_code char(5), primary key (address_id) );	create table propadd (address_id char(5), street char(40), area char(40), city char(40), state char(2), zip_code char(5));
Property_log	PropLog.dbf
create table userid.property_log (property_id char(5) not null , download_timestamp timestamp, last_update timestamp, status char(15) not null , primary key (property_id) );	create table proplog (prop_id char(5), download char(25) , update char(25) , status char(15));

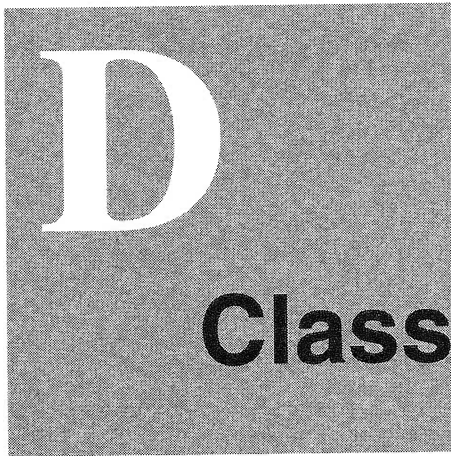
**Table 77.** Database Layout for the REAL Database

DB2 Table/View	Dbf File
Sale_transaction	Sale.dbf
create table userid.sale_transaction (transaction_id timestamp not null , last_update timestamp not null , agreement_form_id char(254), status char(10) not null , buyer_id char(11) not null , property_id char(5) not null , primary key (transaction_id) );	create table sale (trans_id char(25), update char(25), form_id char(254), status char(10), buyer_id char(11), prop_id char(5));
Buyer_info view	
create view userid.buyer_info (buyer_id, first_name, last_name, income, work_phone, home_phone, street, area, city, state, zip_code, max_price, min_price, max_size, min_size, bedrooms, bathrooms, stories, heating, cooling) as select a.buyer_id, first_name, last_name, income, work_phone, home_phone, street, area, city, state, zip_code, max_price, min_price, max_size, min_size, bedrooms, bathrooms, stories, heating, cooling from userid.buyer a, userid.buyer_address b, userid.preference c where (a.buyer_id=address_id and a.buyer_id=c.buyer_id);	

**Table 77. Database Layout for the REAL Database**

DB2 Table/View	Dbf File
<b>Prop_ad_log view</b>	
<pre>create view userid.prop_ad_log (property_id, size,  bedrooms,    bathrooms,  area,        city,  state,       status,  price,       commission_rate,  down_payment_rate) as select a.property_id, size,  bedrooms,    bathrooms,  area,        city,  state,       status,  price,       commission_rate,  down_payment_rate from userid.property a,      userid.property_address b,      userid.property_log c,      userid.marketing_info d where (a.property_id=address_id and  a.property_id=c.property_id and  a.property_id=d.property_id);</pre>	
<b>List_area view</b>	
<pre>create view userid.list_area (area) as select distinct area from userid.property_address a,      userid.property_log b where (address_id = property_id) and       (status = 'available');</pre>	





# **Class Dictionary**

In this appendix we list the parts that you need to build the sample application.

## Visual Parts

Part Name	VBB File	File Stem	Description
AAddressView	VRCOMM	vrcadv	Address view of buyer and property
ADeleteDialogView	VRCOMM	vrcdel	Warning delete dialog view
ALogonView	VRCOMM	vrclogv	Logon view to connect to the data-base
AResultSettingsView	VRCOMM	vrcrsetnv	Application settings view
APropertyCreateView	VRPROP	vrpctv	View for recording property
APropertyManagement-View	VRPROP	vrpmngv	Primary view of the property sub-system
APropertySearchParameterView	VRPROP	vrpsrcv	Search criteria view for property
APropertySearchResult-View	VRPROP	vrpsrrsv	Tabular view of properties
APropertyUpdateView	VRPROP	vrpupdv	View for updating property
APropertyView	VRPROP	vrpprv	Property view
AUploadView	VRSERV	vrsuplv	Dialog box for generating the export files
ASimulMortgageView	VRSERV	vrssimv	Dialog box for gathering buyer's mortgage information
ASimulMortgageFrame-View	VRSERV	vrssimf	Frame window to present ASimul-MortgageView
AResultMainView	VRMAIN	vrmain	Main application view

## Nonvisual Parts

Part Name	VBB File	File Stem	Description
NumDecOnlyKbdHandler	KBDHDR	kbdhdr	General-purpose keyboard handler
NumOnlyKbdHandler	KBDHDR	kbdhdr	General-purpose keyboard handler
AMortgageCalculator	VRSERV	vrssimc	Estimate a mortgage
AMarketingInfo	VRPROP	vrpmrkt	Calculate property derived attributes
List_area	VRPROP	vrpctv	DACS part of LIST_AREA view
List_areaManager	VRPROP	vrpmngv	VDACS part of LIST_AREA view

Part Name	VBB File	File Stem	Description
Marketing_info	VRPROP	vrpsrcv	DACS part of MARKETING_INFO table
Marketing_infoManager	VRPROP	vrpsrrsv	DACS part of MARKETING_INFO table
Multidoc	VRPROP	vrpsrrsv	DACS part of MULTIDOC table
MultidocManager	VRPROP	vrpsrrsv	DACS part of MULTIDOC table
Property	VRPROP	vrpsrrsv	DACS part of PROPERTY table
PropertyManager	VRPROP	vrpsrrsv	DACS part of PROPERTY table
Property_address	VRPROP	vrpsrrsv	DACS part of PROPERTY_ADDRESS table
Property_addressManager	VRPROP	vrpsrrsv	DACS part of PROPERTY_ADDRESS table
Property_log	VRPROP	vrpsrrsv	DACS part of PROPERTY_LOG table
Property_logManager	VRPROP	vrpsrrsv	DACS part of PROPERTY_LOG table
Property_ad_log	VRPROP	vrpsrrsv	DACS part of PROP_AD_LOG view
Property_ad_logManager	VRPROP	vrpsrrsv	DACS part of PROP_AD_LOG view





# E

## Source Listings

In this appendix we provide the source code for the `BuildClause` member function of `APropertySearchParameterView` (see Figure 172 on page 526 ). We also give the source code for the `FlatFile` class (see Figure 173 on page 526 and Figure 174 on page 527 ).

## BuildClause member function

```

//*****
//
// Declaration of the BuildClause member function
//
// Description:
// This method builds an SQL clause from entry controls.
// The inputs are selected according to the status of
// APropertySearchParameterView check boxes.
//
//*****
public:
    Boolean APropertySearchParameterView::BuildClause();

```

**Figure 172.** BuildClause Member Function: Declaration

## Flat File Class

```

//*****
// FILE NAME: FLATFILE.h
//
// DESCRIPTION:
// Constant declarations for class:
// FlatFile- This part handles a file
// -----
// Warning: This file was generated by the VisualAge C++
// Visual Builder.
// Modifications to this source file will be lost when the part
// is regenerated.
//*****
#ifndef _ICCONST_
#include <icconst.h>
#endif
#ifndef _IVBDEFS_
#include <ivbdefs.h>
#endif
#ifndef WND_FlatFile
#define WND_FlatFile VBBASEWINDOWID
#endif

```

**Figure 173.** Flat File Class: H File

```

/***** FILE NAME: FLATFILE.hpp *****/
//
// DESCRIPTION:
// Declaration of the class:
// FlatFile- This part handles a file
// -----
// Warning: This file was generated by the VisualAge C++
// Visual Builder.
// Modifications to this source file will be lost when the part
// is regenerated.
// *****/
#ifndef _FLATFILE_
#define _FLATFILE_
class FlatFile;
#ifndef _ISTDNTFY_
#include <istdntfy.hpp>
#endif
#include "istring.hpp"
#include "fstream.h"
#include "FLATFILE.h"

//-----
// Align classes on four-byte boundary.
//-----
#pragma pack(4)
/*****
// Class definition for FlatFile
*****/
class FlatFile : public IStandardNotifier {
public:
    //-----
    // Constructors / destructors
    //-----

```

**Figure 174.** (Part 1 of 2) Flat File Class: HPP File

```
FlatFile();
virtual ~FlatFile();
//-----
// public member functions
//-----
virtual FlatFile & initializePart();
// public member data
//-----
static const INotificationId readyId;
protected:
//-----
// protected member functions
//-----
virtual Boolean makeConnections();
private:
#include "flatfile.hpv"
}; //FlatFile

//-----
// Resume compiler default packing.
//-----
#pragma pack()
#endif
```

**Figure 174.** (Part 2 of 2) Flat File Class: HPP File

```

/***** FILE NAME: FLATFILE.cpp *****/
//
// DESCRIPTION:
//   Class implementation of the class:
//   FlatFile- This part handles a file
// -----
// Warning: This file was generated by the VisualAge C++
// Visual Builder.
// Modifications to this source file will be lost when the part
// regenerated.
/*****
#ifndef _INOTIFEV_
#include <inotifev.hpp>#endif

#ifndef _IOBSERVER_
#include <iobserver.hpp>#endif

#ifndef _ISTDNTFY_
#include <istdntfy.hpp>
#endif
#ifndef _FLATFILE_
#include "FLATFILE.HPP"
#endif
#ifndef _IVBDEFS_
#include <ivbdefs.h>
#endif
#ifndef _ITRACE_
#include <itrace.hpp>
#endif

#pragma export (FlatFile::readyId)
const INotificationId FlatFile::readyId = "FlatFile::readyId";
//-----
// FlatFile :: FlatFile
//-----

```

**Figure 175.** (Part 1 of 2) Flat File Class: CPP File

```
#pragma export (FlatFile::FlatFile())
FlatFile::FlatFile()
{
} //end constructor
//-----
// FlatFile :: ~FlatFile
//-----
#pragma export (FlatFile::~~FlatFile())
FlatFile::~~FlatFile()

}
//-----
// FlatFile :: initializePart
//-----
#pragma export (FlatFile::initializePart())
FlatFile & FlatFile::initializePart()
{
    makeConnections();
    notifyObservers(INotificationEvent(readyId, *this));
    return *this;
}

//-----
// FlatFile :: makeConnections
//-----
#pragma export (FlatFile::makeConnections())
Boolean FlatFile::makeConnections()
{
    this->enableNotification();
    return true;
}

#include "flatfile.cpv"
```

**Figure 175.** (Part 2 of 2) Flat File Class: CPP File

```
// Default Part Code Generation begins here...
public:
    FlatFile& close();
    FlatFile& open( IString newFileName );
    FlatFile& readLine();
    FlatFile& readFile();
    FlatFile& setCurrentLine(const IString& aNewLine);
    FlatFile& writeLine(IString aLine);
    FlatFile& setFileName(const IString& aFileName);
    IString fileName() const;
    IString currentLine() const;
    static INotificationId currentLineId;
    static INotificationId endOfFileId;
    static INotificationId openedId;
private:
    IString iFileName;
    IString iCurrentLine;
    fstream aFile;
    Boolean eofReached;
    Boolean aFileIsOpen;
// Default Part Code Generation ends here.
```

**Figure 176.** Flat File Class: HPV File



```
// Default Part Code Generation begins here...
INotificationId FlatFile::openedId = "FlatFile::opened";
INotificationId FlatFile::endOfFileId = "FlatFile::endOfFile";
INotificationId FlatFile::currentLineId = "FlatFile::currentLine";

IString FlatFile::fileName() const
{
    IFUNCTRACE_DEVELOP();
    return iFileName;
}

FlatFile& FlatFile::setFileName(const IString& aFileName)
{
    IFUNCTRACE_DEVELOP();
    if (iFileName != aFileName)
    {
        iFileName = aFileName;
    }
    return *this;
}

IString FlatFile::currentLine() const
{
    IFUNCTRACE_DEVELOP();
    return iCurrentLine;
}

FlatFile& FlatFile::setCurrentLine(const IString& aNewLine)
{
    IFUNCTRACE_DEVELOP();
    iCurrentLine = aNewLine;
    ITRACE_DEVELOP( "NewLine ->" + iCurrentLine );
    notifyObservers(INotificationEvent(FlatFile::currentLineId, *this))
;
    return *this;
}

FlatFile& FlatFile::readLine()
{
    IFUNCTRACE_DEVELOP();
```

**Figure 177.** (Part 1 of 3) Flat File Class: CPV File

```

    IString strNewLine = IString::lineFrom( aFile );
    ITRACE_DEVELOP( "Line -> " + strNewLine );
    if( aFile.fail() )
    {
        eofReached = true;
        notifyObservers(INotificationEvent(FlatFile::endOfFileId, *this));
    }
    else
    {
        setCurrentLine( strNewLine );
    }
    return *this;
}

FlatFile& FlatFile::readFile()
{
    IFUNCTRACE_DEVELOP();
    while( eofReached == false )
    {
        readLine();
    } /* endwhile */
    return *this;
}

// be aware that this function writes at the end of file only
FlatFile& FlatFile::writeLine(IString aLine)
{
    IFUNCTRACE_DEVELOP();
    ITRACE_DEVELOP( "Writing -> " + aLine );
    // save read pointer
    long mark = aFile.tellg();
    // write line ( always to end of file )
    aLine = aLine + "\n";
    aFile << aLine;
    // restore read pointer
    aFile.seekg( mark );
    return *this;
}

```

**Figure 177.** (Part 2 of 3) Flat File Class: CPV File

```

{
    IFUNCTRACE_DEVELOP();
    if( aFileIsOpen )
    {
        close();
    } /* endif */
    ITRACE_DEVELOP( "Opening-> " + newFileName );
    // open file for read and append
    aFile.open( newFileName, ios::in | ios::app );
    if( aFile.fail() == 0 )
    {
        eofReached = false;
        aFileIsOpen = true;
        iFileName = newFileName;
        // reset file position to beginning of file
        aFile.seekg( 0, ios::beg );
        notifyObservers(INotificationEvent(FlatFile::openedId, *this));
    } /* endif */
    else
    {
        IAccessError exc = IAccessError( "Could not open file: " +
                                         iFileName );
        ITHROW( exc );
    }
    return *this;
}

FlatFile& FlatFile::close()
{
    IFUNCTRACE_DEVELOP();
    if( aFileIsOpen )
    {
        aFile.close();
        aFileIsOpen = false;
        iFileName = " ";
    } /* endif */
    return *this;
}

// Feature source code generation begins here...

// Feature source code generation ends here.

```

**Figure 177.** (Part 3 of 3) Flat File Class: CPV File

---

# Glossary

This glossary defines terms and abbreviations that are used in this book. If you do not find the term you are looking for, refer to the *IBM Dictionary of Computing*, New York:McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

## A

**abstract class.** A class that provides common behavior across a set of subclasses but is not itself designed to have instances that work. An abstract class represents a concept; classes derived from it represent implementations of the concept. For example, *IControl* is the abstract base class for control view windows; the *ICanvas* and *IListBox* classes are controls derived from *IControl*. An abstract class must have at least one pure virtual function.

See also *base class*.

**access.** A property of a class that determines whether a class member is accessible in an expression or declaration.

**action.** A specification of a function that a part can perform. The Visual Builder uses action specifications to generate connections between parts. Actions are resolved to member function calls in the generated code.

Compare to *attribute* and *event*.

**argument.** A data element, or value, included as part of a member function call. Arguments provide additional information that the called member function can use to perform the requested operation.

**attribute.** A specification of a property of a part. For example, a customer part could have a name attribute and an address attribute. An attribute can itself be a part with its own behavior and attributes.

The Visual Builder uses attribute specifications to generate code to get and set part properties.

Compare to *action* and *event*.

**attribute-to-action connection.** A connection that starts an action whenever an attribute's value changes. It is similar to an event-to-action connection because the attribute's event ID is used to notify the action when the value of the attribute changes.

See also *connection*. Compare to *event-to-action connection*.

**attribute-to-attribute connection.** A connection from an attribute of one part to an attribute of another part. When one attribute is updated, the other attribute is updated automatically.

See also *connection*.

**attribute-to-member function connection.** A connection from an attribute of a part to a member function. When the attribute undergoes a state change, then the member function is called.

See also *connection*.

---

## B

**base class.** A class from which other classes or parts are derived. A base class may itself be derived from another base class.

See also *abstract class*.

**behavior.** The set of external characteristics that an object exhibits.

**breakpoint.** A point in a computer program where the execution may be halted.

**build.** An action that invokes the Work-Frame Build tool. The Build tool managed the project's make file, as well as build dependencies between project in a project hierarchy.

## C

**call level interface (CLI).** A callable application program interface (API) for database access, which is an alternative to an embedded SQL application program interface. In contrast to embedded SQL, CLI does not require precompiling or binding by the user, but instead provides a standard set of functions to process SQL statements and related services at run time.

**caller.** An object that sends a member function call to another object.

Contrast with *receiver*.

**canvas.** Canvases are windows with a layout algorithm that manages child windows. The canvas classes are a set of window classes that allow you to implement dialog boxes. The different canvases can manage the size and position of child windows, provide moveable split bars between windows, and support the ability to scroll a window. The canvases include the base class, *ICanvas*, and its four subclasses: *IMulticellCanvas*, *ISetCanvas*, *ISplitCanvas*, and *IViewPort*.

**category.** In the Composition Editor, a selectable grouping of parts represented by an icon in the leftmost column. Selecting a category displays the parts belonging to that category in the next column.

See also *parts palette*.

**CDF.** See *Compound Document Framework*.

**class.** An aggregate that can contain functions, types, and user-defined operators, in addition to data. Classes can be defined hierarchically, allowing one class to be an expansion of another, and can restrict access to its members.

**Class Editor.** The editor used to specify the names of files that Visual Builder writes to when the user generates default code. This editor can also be used to do the following:

- ☐ Enter a description of the part
- ☐ Specify a different .vbb file in which to store the part
- ☐ See the name of the part's base class
- ☐ Modify the part's default constructor
- ☐ Enter additional constructor and destructor code
- ☐ Specify a .lib file for the part
- ☐ Specify a resource DLL and ID to assign an icon to the part
- ☐ Specify other files to be included when the application is built.

Compare to *Composition Editor* and *Part Interface Editor*.

**class hierarchy.** A tree-like structure showing relationships among object classes. It places one abstract class at the top (a base class) and one or more layers of less abstract classes below it.

**class identifier (CLSID).** In Windows95 and Windows NT, the globally unique identifier for an object. The Win32 system registration database uses CLSIDs to distinguish all OLE objects available on a system. Also commonly referred to as a globally unique identifier (GUID).

**class library.** A collection of classes.

---

**class member function.** See *member function*.

**CLI.** See *Call Level Interface*

**client area object.** An intermediate window between a frame window (IFrameWindow) and its controls and other child windows.

**client object.** An object that requests services from other objects.

**collection.** A set of features in which each feature is an object.

**Collection Class Library.** A C++ class library that provide basic functions for collections, and can be used as base classes.

**COM.** See *Compound Object Model*.

**Common Object Request Broker Architecture (CORBA).**

**Common User Access (CUA).** An IBM architecture for designing graphical user interfaces by use of a set of standard components and terminology.

**Complex Mathematics Library.** A C++ class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

**composite part.** A part that is composed of a part and one or more subparts. A composite part can contain visual parts, nonvisual parts, or both.

See also *nonvisual part*, *part*, *subpart*, and *visual part*.

**Composition Editor.** A view that is used to build a graphical user interface and to make connections between parts.

Compare to *Class Editor* and *Part Interface Editor*.

**compound document.** A means for integrating arbitrary or unstructured data from different sources into one centralized location.

**Compound Document Framework.** A starting point for creating a server a container document that is OLE-enabled. The framework stores compound documents using the OLE-structured storage specification (docfiles).

**Compound Object Model (COM).** The underlying model for all OLE services. It consists of a variety of APIs and object interfaces that allow container components to communicate and interact with one another.

**concrete class.** A subclass of an abstract class that is a specialization of the abstract class.

**connection.** A formal, explicit relationship between parts. Making connections is the basic technique for building any visual application because that defines the way in which parts communicate with one another. The visual builder generates the code that then implements these connections.

See also *attribute-to-action connection*, *attribute-to-attribute connection*, *attribute-to-member function connection*, *custom logic connection*, *event-to-action connection*, *event-to-attribute connection*, *event-to-member function connection*, and *parameter connection*.

**const.** An attribute of a data object that declares that the object cannot be changed.

**construction from parts.** A software development technology in which applications are assembled from existing and reusable software components, known as parts.

**constructor.** A special class member function that has the same name as the class and is used to construct and possibly initialize objects of its class type.

**CUA.** See *Common User Access*.

**cursored emphasis.** The appellation of a choice when the selection cursor is on that choice.

---

**custom logic connection.** A connection that causes customized C or C++ code to be run. This connection can be triggered either when an attribute's value changes or when an event occurs.

## D

**data abstraction.** A data type with a private representation and a public set of operations. The C++ language uses the concept of classes to implement data abstraction.

**database.** (1) A systematized collection of data that can be accessed and operated upon by an information processing system. (2) A collection of information such as tables, views, and indexes.

**database management system (DBMS).** A computer program that manages data by providing the services of centralized control, data independence, and complex physical structures for efficient access, integrity, recovery, concurrency control, privacy, and security.

**data member.** Private data that belongs to a given object and is hidden from direct access by all other objects. Data members can only be accessed by the member functions of the defining class and its subclasses.

**data model.** A combination of the base classes and parts shipped with the product and the classes and parts a user saves and creates. They are saved in a file named vbbase.vbb.

**data object.** A storage area used to hold a value.

**DB2 Call Level Interface (CLI).** The DB2 call level interface is an alternative SQL interface for the DB2 family of products and takes full advantage of DB2 capability. This implementation closely follows industry standards, such as X/OPEN™, to enhance application portability. Currently, the DB2 Call Level Interface functions are compatible with ODBC 2.0, and contain DB2 specific APIs to help exploit DB2 capability.

DB2 for MVS/ESA. An IBM relational database management system for the MVS operating system.

**DBCS.** See double-byte character set.

**DBMS.** See *database management system*.

**declaration.** A description that makes an external object or function available to a function or a block.

**DEF file.** See *module definition file*.

**derivation.** The creation of a new or abstract class from an existing or base class.

**derived class.** A class that inherits from a base class. You can add new data members and members functions to the derived class. You can manipulate a derived class object as if it were a base class object. The derived class can override virtual functions of the base class.

Synonym for *child class* and *subclass*.

**destructor.** A special class member function that has the same name as the class and is used to destroy class objects.

**DLL.** See *dynamic link library*.

**double-byte character set (DBCS).** A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols that can be represented by 256 code points, require double-byte character sets.

Compare to *SBCS*.

**dynamic link library (DLL).** A library containing data and code objects that can be used by programs or applications during loading or at run time. Although they are not part of the program's executable (.exe) file, they are required for an .exe file to run properly.

---

## E

**encapsulation.** The hiding of a software object's internal representation. The object provides an interface that queries and manipulates the data without exposing its underlying structure.

**event.** A specification of a notification from a part.

Compare to *action*, *attribute*, and *part event*.

**event-to-action connection.** A connection that causes an action to be performed when an event occurs.

See also *connection*.

**event-to-attribute connection.** A connection that changes the value of an attribute when a certain event occurs.

See also *connection*.

**event-to-member function connection.** A connection from an event of a part to a member function. When the connected event occurs, the member function is executed.

See also *connection*.

**expansion area.** The section of a multicell canvas between the current cell grid and the outer edge of the canvas. Visually, this area is bounded by the rightmost column gridline and the bottommost row gridline.

## F

**feature.** (1) A major component of a software product that can be installed separately. (2) In Visual Builder, an action, attribute, or event that is available from a part's part interface and that other parts can connect to.

**full attribute.** An attribute that has all of the behaviors and characteristics that an attribute can have: a data member, a get member function, a set member function, and an event identifier.

**free-form surface.** The large open area of the Composition Editor window. The free-form surface holds the visual parts contained in the views a user builds and representations of the nonvisual parts (models) that an application includes.

**framework.** A set of cooperative classes with strong connections that provide a template for development.

See also *compound document framework* and *notification framework*.

Compare to *class library*.

## G

**graphical user interface (GUI).** A type of interface that enables users to communicate with a program by manipulating graphical features, rather than by entering commands. Typically, a graphical user interface includes a combination of graphics, pointing devices, menu bars and other menus, overlapping windows, and icons.

**GUI.** See *graphical user interface*.

## H

**handles.** Small squares that appear on the corners of a selected visual part in the visual builder. Handles are used to resize parts.

Compare to *primary selection*.

**header file.** A file that contains system-defined control information that precedes user data.

**heap storage.** An area of storage used for allocation of storage whose lifetime is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments.



---

## K

**keyword.** A predefined word reserved for the C and C++ languages, that may not be used as an identifier.

## I

**inheritance.** (1) A mechanism by which an object class can use the attributes, relationships, and member functions defined in more abstract classes related to it (its base classes). (2) An object-oriented programming technique that allows one to use existing classes as bases for creating other classes.

**instance.** Synonym for *object*, a particular instantiation of a data type.

**interface definition language (IDL).** The formal language (independent from any programming language) by which the interface for a class of objects is defined in a .idl file, which the SOM Compiler then interprets to create an implementation template file and bindings files. SOM's IDL is fully compliant with the CORBA standard established by the OMG.

**interface repository (IR).** The database that SOM optionally creates to provide persistent storage of objects representing the major elements of interface definitions. Creation and maintenance of the IR is based on information supplied in the IDL source file.

## L

**legacy code.** Existing code that a user might have. Legacy applications often have character-based, nongraphical user interfaces; usually they are written in a non-object-oriented language, such as C or COBOL.

**loaded.** The state of the mouse pointer between the time one selects a part from the parts palette and deposits the part on the free-form surface.

**locale.** The definition of the subset of a user's environment that depends on language and cultural conventions.

## M

**main part.** The part that users see when they start an application. This is the part from which the `main()` function C++ code for the application is generated. The main part is a special kind of composite part.

See also *part* and *subpart*.

**mangling.** The encoding during compilation of identifiers such as function and variable names to include type and scope information. This information is stored in object files and executables. The prelinker uses this information to ensure type-safe linkage.

**member.** (1) A data object in a structure or a union. (2) In C++, classes and structures can also contain functions and types as members.

**member function.** An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class.

**member function call.** A communication from one object to another that requests the receiving object to execute a member function.

A member function call consists of a member function name that indicates the requested member function and the arguments to be used in executing the member function. The member function call always returns some object to the requesting object as the result of performing the member function.

Synonym for *message*.

**member function name.** The component of a member function call that specifies the requested operation.

---

**message.** A request from one object that the receiving object implement a member function. Because data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name of the receiving object, the member function to be implemented, and any arguments the member function needs for implementation.

Synonym for *member function call*.

**model.** A nonvisual part that represents the state and behavior of an object, such as a customer or an account.

Contrast with *view*.

**module definition file.** A file that describes the code segments within a load module.

Synonym for *DEF file*.

## N

**nested class.** A class defined within the scope of another class.

**nonvisual part.** A part that has no visual representation at run time. A nonvisual part typically represents some real-world object that exists in the business environment.

Compare to *model*. Contrast with *view* and *visual part*.

**no-event attribute.** An attribute that does not have an event identifier.

**no-set attribute.** An attribute that does not have a set member function.

**notebook part.** A visual part that resembles a bound notebook containing pages separated into sections by tabbed divider pages. A user can turn the pages of a notebook or select the tabs to move from one section to another.

**notification framework.** A set of classes that implement the *notifier/observer* protocol. The notification framework is the base of the construction from parts technology (Visual Builder).

## O

**object.** (1) A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon. (2) A collection of data and member functions that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and member functions. Each object in the class is said to be an instance of the class. (3) An instance of an object class consisting of attributes, a data structure, and operational member functions. It can represent a person, place, thing, event, or concept. Each instance has the same properties, attributes, and member functions as other instances of the object class, though it has unique values assigned to its attributes.

**object linking and embedding (OLE).** (1) An API that supports compound documents, cross-application macro control, and common object registration. OLE defines protocols for in-place editing, drag-and-drop data transfers, structured storage, custom controls, and more. (2) A data-sharing scheme that allows dissimilar applications to create complex documents cooperatively. The documents can consist of material that a single application could not have created on its own.

**object class.** A template for defining the attributes and member functions of an object. An object class can contain other object classes. An individual representation of an object class is called an object.

**object factory.** A nonvisual part capable of dynamically creating new instances of a specified part. For example, during the execution of an application, an object factory can create instances of a new class to collect the data being generated.

---

**object-oriented programming (OOP).** A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on those data objects that constitute the problem and how they are manipulated, not on how something is accomplished.

**observer.** An object that receives notification from a notifier object.

**ODBC.** See *Open Database Connectivity*.

**ODBC Driver.** An ODBC driver is a dynamically-linked library (DLL) that implements ODBC function calls and interacts with a data source.

**ODBC Driver Manager.** The ODBC driver manager, provided by Microsoft™, is a DLL with an import library. The primary purpose of the Driver Manager is to load ODBC drivers. The Driver Manager also:

- ❑ provides entry points to ODBC functions for each driver
- ❑ provides parameter validation and sequence validation for ODBC calls.

**OLE.** See *Object Linking and Embedding*.

**Open Database Connectivity (ODBC).** A Microsoft™ developed C database application programming interface (API) that allows access to database management systems calling callable SQL, which does not require the use of a SQL preprocessor. In addition, ODBC provides an architecture which allows users to add modules called database drivers that link the application to their choice of database management systems at run time. This means applications no longer need to be directly linked to the modules of all the database management systems that are supported.

**operation.** A member function or service that can be requested of an object.

**Object Request Broker (ORB).** A CORBA term designating the means by which objects transparently make requests and receive responses from objects, whether they are local or remote.

**overloading.** An object-oriented programming technique that allows redefinition of functions and most standard C++ operators when the functions and operators are used with class types.

## P

**palette.** See *parts palette*.

**parameter connection.** A connection that satisfies a parameter of an action or member function by supplying either an attribute's value or the return value of an action, member function, or custom logic. The parameter is always the source of the connection.

See also *connection*.

**parent class.** The class from which another part or class inherits data, member functions, or both.

**part.** A self-contained software object with a standardized public interface, consisting of a set of external features that allow the part to interact with other parts. A part is implemented as a class that supports the INotifier protocol and has a part interface defined.

The parts on the palette can be used as templates to create instances or objects.

**part event.** A representation of a change that occurs to a part. The events on a part's interface enable other interested parts to receive notification when something about the part changes. For example, a push button generates an event signaling that it has been clicked, which might cause another part to display a window.

**part event ID.** The name of a part static-data member used to identify the notification that is being signaled.

**part interface.** A set of external features that allows a part to interact with other parts. A part's interface is made up of three characteristics: attributes, actions, and events.

---

**Part Interface Editor.** An editor that the application developer uses to create and modify attributes, actions, and events, which together make up the interface of a part.

Compare to *Class Editor* and *Composition Editor*.

**parts palette.** A palette control that holds a collection of visual and nonvisual parts used in building additional parts for an application. The parts palette is organized into *categories*. Application developers can add parts to the palette for use in defining applications or other parts.

**pragma.** An implementation-defined instruction to the compiler.

**preferred features.** A subset of the part's features that appear in a pop-up connection menu. Generally, they are the features used most often.

**primary selection.** In the Composition Editor, the part used as a base for an action that affects several parts. For example, an alignment tool will align all selected parts with the primary selection. Primary selection is indicated by closed (solid) selection handles, whereas the other selected parts have open selection handles.

See also *selection handles*.

**promote features.** Make features of a subpart available to be used for making connections. This applies to subparts that are to be included in other parts, for example, a subpart consisting of three push buttons on a canvas. If this example subpart is placed in a frame window, the features of the push buttons would have to be promoted to make them available from within the frame window.

**private.** Pertaining to a class member that is accessible only to member functions and friends of that class.

**process.** A program running under OS/2, along with the resources associated with it (memory, threads, file system resources, and so on).

**program.** (1) One or more files containing a set of instructions conforming to a particular programming language syntax. (2) A self-contained, executable module. Multiple copies of the same program can be run in different processes.

**protected.** Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

**prototype.** A function declaration or definition that includes both the return type of the function and the types of its arguments.

**primitive part.** A basic building block of other parts. A primitive part can be relatively complex in terms of the function it provides.

**process.** A collection of code, data, and other system resources, including at least one thread of execution, that performs a data processing task.

**property.** A unique characteristic of a part.

**pure virtual function.** A virtual function that has a function definition of = 0;.

## R

**receiver.** The object that receives a member function call.

Contrast with *caller*.

**resource file.** A file that contains data used by an application, such as text strings and icons.

**runtime type information (RTTI).** An extension to the C++ language that allows to retrieve the type of an object at run time, and to perform safe casting operations.

---

## S

**SBCS.** See *single-byte character set*

**selection handles.** In the Composition Editor, small squares that appear on the corners of a selected visual part. Selection handles are used to resize parts.

See also *primary selection*.

**server.** A computer that provides services to multiple users or workstations in a network; for example, a file server, a print server, or a mail server.

**service.** A specific behavior that an object is responsible for exhibiting.

**settings view.** A view of a part that provides a way to display and set the attributes and options associated with the part.

**single-byte character set.** A set of characters in which each character is represented by a 1- byte code.

**sticky.** In the Composition Editor, the mode that enables an application developer to add multiple parts of the same class (for example, three push buttons) without going back and forth between the parts palette and the free-form surface.

**structure.** A construct that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types.

**subpart.** A part that is used to create another part.

See also *nonvisual part*, *part*, and *visual part*.

**superclass.** See *abstract class* and *base class*.

## T

**tear-off attribute.** An attribute that an application developer has exposed to work with as though it were a stand-alone part.

**template.** A family of classes or functions where the code remains invariant but operates with variable types.

**template class.** A class instance generated by a class template.

**template function.** A function generated by a function template.

**thread.** A unit of execution within a process.

**tic.** Approximately 838 nanoseconds. A tic is the smallest unit of time measured by the Performance Analyzer counter.

**tool bar.** The strip of icons along the top of the free-form surface. The toolbar contains tools to help an application developer construct composite parts.

**type class.** In WorkFrame, represents the method by which an object is determined to be a member of a type. "File Mask" is an example of a type class. Membership to a "file mask" type is determined by matching the file mask filter to the object's name.

## U

**UI.** See *user interface*.

**unloaded.** The state of the mouse pointer before a user selects a part from the parts palette and after the user deposits a part on the free-form surface. In addition, a user can unload the mouse pointer by pressing the Esc key.

**user interface (UI).** (1) The hardware, software, or both that enables a user to interact with a computer. (2) The term *user interface* normally refers to the visual presentation and its underlying software with which a user interacts.

---

## V

**variable.** (1) A storage place within an object for a data feature. The data feature is an object, such as number or date, stored as an attribute of the containing object. (2) A part that receives an identity at run time. A variable by itself contains no data or program logic; it must be connected such that it receives run-time identity from a part elsewhere in the application.

**view.** (1) A visual part, such as a window, push button, or entry field. (2) A visual representation that can display and change the underlying model objects of an application. Views are both the end result of developing an application and the basic unit of composition of user interfaces.

Compare to *visual part*. Contrast with *model*.

**virtual function.** A function of a class that is declared with the keyword *virtual*. The implementation that is executed when a program makes a call to a virtual function depends on the type of the object for which it is called. This is determined at run time.

**visual part.** A part that has a visual representation at run time. Visual parts, such as windows, push buttons, and entry fields, make up the user interface of an application.

Compare to *view*. Contrast with *nonvisual part*.

**visual programming tool.** A tool that provides a means for specifying programs graphically. Application programmers write applications by manipulating graphical representations of components.

## W

**white space.** Space characters, tab characters, form-feed characters, and new-line characters.

**window.** (1) A rectangular area of the screen with visible boundaries in which information is displayed. Windows can overlap on the screen, giving it the appearance of one window being on top of another. (2) In the Composition Editor, a part that can be used as a container for other visual parts, such as push buttons.



---

# List of Abbreviations

<b>ANSI</b>	American National Standards Institute	<b>IDE</b>	integrated development environment
<b>APA</b>	all points addressable	<b>IDL</b>	interface definition language
<b>API</b>	application programming interface	<b>IOC</b>	IBM Open Class Library
<b>CAD</b>	computer-aided design	<b>ISO</b>	International Organization for Standardization
<b>CAE</b>	Client Access Enabler	<b>ITSO</b>	International Technical Support Organization
<b>CDF</b>	Compound Document Framework	<b>IWO</b>	IBM Workframe option project
<b>CLI</b>	call level interface	<b>IWP</b>	IBM Workframe project
<b>CLSID</b>	class identifier	<b>LAN</b>	local area network
<b>COFF</b>	common object file format	<b>LPEX</b>	Live Parsing EXtensible editor
<b>CORBA</b>	Common Object Request Broker Architecture	<b>MMPM/2</b>	Multimedia Presentation Manager/2
<b>CRC</b>	class-responsibility-collaborator	<b>MFC</b>	Microsoft Foundation Classes
<b>DB2</b>	DATABASE 2	<b>NBF</b>	NetBIOS frame
<b>DBF</b>	dBase file	<b>NBT</b>	NetBIOS over TCP/IP
<b>DBMS</b>	database management system	<b>NetBIOS</b>	network basic input/output system
<b>DDCS</b>	distributed database connection services	<b>NLS</b>	National Language Support
<b>DDE</b>	dynamic data exchange	<b>NTSF</b>	Windows NT™ file system
<b>DIS</b>	draft international standard	<b>ODBC</b>	Open Database Connectivity
<b>DLL</b>	dynamic link library	<b>OLE</b>	Object Linking and Embedding
<b>DNS</b>	domain name system	<b>OMF</b>	object module format
<b>DRDA</b>	Distributed Relational Database Architecture	<b>OMG</b>	Object Management Group
<b>DSOM</b>	Distributed System Object Model	<b>OMT</b>	object modeling technique
<b>DTS</b>	Direct-to-SOM	<b>OOA</b>	object-oriented analysis
<b>FAT</b>	File Allocation Table	<b>OOD</b>	object-oriented design
<b>GUI</b>	graphical user interface	<b>OOSE</b>	object-oriented software engineering
<b>HPFS</b>	high-performance file system	<b>PM</b>	Presentation Manager
<b>IBM</b>	International Business Machines Corporation	<b>RAD</b>	rapid application development



---

<b><i>RBBC</i></b>	release-to-release binary compatibility
<b><i>RDD</i></b>	responsibility-driven design
<b><i>RTF</i></b>	rich text format
<b><i>RTTI</i></b>	run-time type identification
<b><i>SDK</i></b>	software developer kit
<b><i>SOM</i></b>	System Object Model
<b><i>SQL</i></b>	Structured Query Language
<b><i>SVGA</i></b>	super video graphics array/adaptor
<b><i>TCP/IP</i></b>	Transmission Control Protocol/Internet Protocol
<b><i>UPM</i></b>	user profile management
<b><i>VBB</i></b>	Visual Builder binary
<b><i>VBE</i></b>	Visual Builder export
<b><i>VMT</i></b>	visual modeling technique
<b><i>WWW</i></b>	World Wide Web

---

# Index

## A

- abstraction 8, 63, 81
- accelerator key 342
- access specifier 283, 285
- actor 67–72
- ANSI 383
- application
  - analysis 5–15, 59–62, 65, 67, 75, 80, 81, 90
  - design 5, 12–15, 60–88, 106, 247, 258, 268, 291
  - embedded 454
  - implementation 12–15, 43, 61, 62, 79, 81, 85, 87, 88, 98, 103, 227, 291
  - problem domain 12–14, 61, 63, 67, 72, 85
  - requirement specification 62, 66, 71, 92, 346
  - tuning 46, 83
- association
  - multiplicity 89
- attribute
  - derived 89, 226, 227, 522
  - tearing off 267, 336, 344, 345, 346

## B

- bind.ddl 382
- boolean part. See IVBBooleanPart
- browser 46
  - database 46, 49, 366
  - interacting with Visual Builder 366
  - QuickBrowse 46, 49, 366
- business object 225, 226
- business part xxix, 16, 90, 103

## C

- call level interface. See CLI 385
- canvas
  - minimum size 153, 154
  - See also IMultiCellCanvas, ISetCanvas, ICanvas
- category

- Buttons 173, 205, 214
- Composers 156, 164, 166, 167, 173
- Data entry 155, 167, 171, 173, 186
- Frame extensions 179, 255, 277
- Lists 143, 156, 194, 198
- Models 261, 267, 276, 295
- Other 252, 261, 270, 276, 348

## CDF

- component 454
- container 454
- document 454
- CD-ROM xxxii, 147, 148, 150
- check box. See ICheckBox
- class 243
  - ancestor/base class 9, 10
  - attribute 7, 8, 9
  - delete 306
  - derived class 9
  - instance 8, 11, 34, 95, 225
  - method 25, 34, 89, 91
- Class Editor 32, 56, 159, 177, 180, 183, 187, 190, 192, 200, 203, 208, 211, 214, 217, 220, 223, 229, 274, 281, 305, 366, 367
- class interface part
  - and notification framework 226
  - See also IDate, ITime
- class-responsability-collaborator card. See CRC card
- CLI 385, 387
- client/server 81, 374, 388
- Code generation
  - User files 33
- code generation file 159, 177, 180, 183, 187, 190, 192, 200, 208, 211, 214, 217, 220, 223, 366
- Code generation files 32
- code trace 53, 282
- coding xxvii, xxxii, 11, 60, 83
- collaborator 12, 73, 74, 84, 259
- collection combination list box. See ICollectionviewComboBox
- collection list box. See ICollectionViewListBox
- combination list box. See IComboBox
- Common Object Request Broker Architecture
  - See CORBA
- compiler
  - code optimization 42
  - DTS 43
  - locales support 42
  - memory management 41
  - precompiled headers 41
  - RTTI 43
  - X/Open standard 42

- composite part 56, 104, 144, 148, 178, 180, 266, 273, 346
  - definition 25
- Composition Editor
  - code generation 32
  - creating a nonvisual part 302, 303
  - free-from surface 16, 29
  - palette 29, 143, 144, 151, 155, 197, 255, 261, 295, 536
  - toolbar 29
- compound document 454
- Compound Document Framework. See CDF
- CONFIG.SYS file 117, 121, 123, 185, 188, 191
- connection
  - attribute-to-attribute 26, 226, 232, 259, 260, 261, 263, 267, 277, 279, 286, 293, 346, 361
  - browsing 264, 301
  - CRC card 247
  - custom logic 227, 261, 271, 272, 273
  - event-to-action 26, 250, 260, 266, 281, 287, 298, 301, 304, 306
  - event-to-attribute 26
  - event-to-custom logic 27
  - event-to-member function 27, 279, 283, 284, 308, 311, 312
  - event-trace 259, 263
  - initialization 27, 358
  - logic connection 283
  - notification framework 355
  - parameter 28, 250, 251, 271, 274, 277, 291, 296, 312
  - reordering 264, 266, 281, 298, 301, 304
  - source 27, 226, 354, 363, 369
  - target 27, 226, 273, 354, 363
  - type 26
- constructor 144, 295, 296, 536
- container. See IVBContainerControl
- copy constructor 144
- CORBA 422
- cppwac2.bnd 382
- CRC card
  - drawing connection 247
- CSet++ 16
- custom logic 393
- C/C++ 16, 37, 144, 271, 282, 283, 353

## D

- DACS. See Data Access Builder
- Data Access Builder 131, 373
  - access method 135

- automate link to application 137, 144
- change class and attribute names 406
- class type 135, 138
- concepts 33
- generated classes 136, 139
- invoke 133
- ODBC conformance level 387
- session 134
- SOM 33
- table mapping 24, 133, 401
- transaction services 33
- Data Access Builder part 139, 390
  - data identifier 140
  - datastore 391, 392
  - key identifier 137
  - used as variable 259, 270, 274
- data identifier. See Data Access Builder part
- data integrity 89, 399
- data source, see ODBC Data Source 386
- data type 7, 89
- database
  - access method 382, 389
  - bind 381, 382, 388
  - connection 269, 270, 276, 330, 339, 375, 388, 391
  - data definition 376, 403, 408, 515
  - data type 398, 400
  - database directory 381
  - directory 134
  - join 408
  - manager 135
  - owner 135, 399
  - physical representation 398
  - primary key 137, 400
  - relational 383
  - security 385
  - server 375
  - table 132, 397, 515
  - type 134
  - view 133, 398, 408, 515
- Database 2. See DB2
- database management
  - deleting rows in table 292, 298
  - retrieving rows from multiple tables 292
  - selecting rows in table 291
- DAX file 134
- DB2
  - authentication 135, 375
  - CLI 141, 387, 390
  - Client Access Enabler 375, 380
  - DATE format 272
  - package name length 137
  - precompile 380, 383, 388, 392, 411
  - Security Server 375

---

- server 374
- Software Developer's Kit 380
- TIME format 272
- DBF file 397, 399, 402, 405, 408, 515
- DDL file 376
- debugger
  - breakpoint management 50
  - memory management 51
- DEF file 432
- definition file
  - See DEF file 432
- design object model 259
- destructor 536
- Direct-to-SOM
  - See DTS
- DLL 45, 51, 113–118, 123, 137, 139, 184, 186, 210, 212, 536
  - browsing 47
  - creating 45
  - debugging 51
  - tracing 53
- DTS 444
- dynamic link library. See DLL
- dynamic memory allocation. See IVBFactory
- dynamic model
  - event-trace diagram 259, 263, 279

## E

- Embedded SQL 135, 141, 381, 383, 388, 390, 392
- encapsulation 10, 258
- entry field. See IEntryField
- environment variable 117, 121, 145, 378, 379
- event 4, 25, 26, 27, 28, 30, 66, 76, 91, 309, 354, 358, 359, 363, 364, 368
- event-driven programming 4
- event-trace diagram
  - connection 259, 263, 279
- exception handling 140
  - showing an exception 282
  - try and catch blocks 282
- exception handling. See IMessageBox

## F

- F1@HIERARCHY 355
- factory object
  - creating part dynamically 104
- factory object. See IVBFactory

- FAT file system 106, 122
- file allocation table file system. See FAT file system
- file selection. See IVBFileDialog
- flat file class
  - converting to a part 363
  - interface 363
- fly over help. See IVBFlyText
- font
  - default 151
  - setting 153, 154, 212

## G

- generalization 9
- graphic push button. See IGraphicPushBut-ton
- graphical user interface. See GUI
- group box. See IGroupBox
- GUI
  - prototype 62, 68, 82, 88

## H

- handler
  - IHandler 237, 238, 245
  - IKeyboardHandler 238
  - NumDecOnlyKbdHandler 226, 238, 243
  - NumOnlyKbdHandler 226, 238, 243
  - using an 237, 245
- help
  - context-sensitive help 178, 261, 271, 346, 347
  - general help 342
- High Performance File System. See HPFS
- hosts file 380
- HPFS 106

## I

- IBRS. See browser
- ICanvas 152, 164, 194, 370
- ICC. See compiler
- ICheckBox 200, 201
- ICLUI 121
- ICollectionViewComboBox 201, 202
- ICollectionViewListBox 143, 201–203
- IComboBox 152, 156, 167
- IComponent 460

---

IComponentStationery 462  
 IComponentStationeryFor 462  
 IContainerColumn 197  
 icon. See IIconControl  
 IC\_TRACE\_DEVELOP 120, 121  
 IDatastore 88, 141–143, 215, 390, 392, 394  
 IDatastoreBase 141, 390  
 IDatastoreDB2 141, 390  
 IDatastoreODBC 141, 390, 392  
 IDATA. See Data Access Builder  
 IDate 272, 273, 333  
 IDE. See WorkFrame  
 IDocumentStorage 461  
 IDSCConnectBaseCanvas 141  
 IDSCConnectCanvas 141  
 IEmbedderModel 459  
 IEntryField 152, 155, 156, 167, 173, 176,  
     189, 201, 205, 217, 220, 354  
 IFlatFileStorage 461  
 IFrameWindow 142, 152, 178, 194, 201, 212,  
     285–297, 363  
 IGraphicPushButton 212  
 IGroupBox 173, 176  
 IGUIBundle 463  
 IHelpWindow 347–349  
 IIconControl 184  
 IInfoArea 178, 179, 182, 207, 214, 221  
 IKeyboardHandler 238  
 ILINK  
 IMenu 255, 294, 341  
 IMenuItem 255, 294, 342  
 IMenuSeparator 255, 342  
 IMessageBox  
     message severity 319  
     showing an exception 282, 361  
     tracing the program flow 318  
 IMMDigitalVideo 253, 290  
 IMMPlayerPanel 171, 173, 254  
 IModel 459  
 implementation phase 15, 16, 227, 291  
 import file. See VBE file  
 IMultiCellCanvas 152, 184, 194, 221  
 IMultiLineEdit 171  
 info area. See InfoArea  
 inheritance 8, 9, 15, 17, 78, 83  
     browsing 46, 47  
 INoteBook 162  
 INotifier 354, 358  
 installing the application 509  
 instance. See class or part instance  
 integrated development environment. See  
     WorkFrame  
 interface definition language. See IDL  
 Internet xxxiii  
 INumericSpinButton 167  
 IObservable 354, 356, 357, 370  
 IPERF. See performance analyzer  
 IPersistentObject 139  
 IPF compiler 347  
 ipoattr.hpp 412  
 IPOManager 140  
 IProfile 218, 325, 333  
 IPushButton 173, 201, 370  
 ISetCanvas 154, 166, 201, 221  
 IStandardNotifier 226, 354, 363  
 IStaticText 155, 167, 172, 175, 186, 210,  
     214, 216, 221, 223  
 IString 401  
 IStringGenerator 202  
 IStructuredStorage 461  
 iterative technique 13, 88, 92  
 ITime 272, 273  
 ITitle 286  
 IVBBooleanPart  
     enabling control 249, 308  
     used with IMessageBox 321, 322  
 IVBContainerControl 192, 201  
 IVBFactory  
     create method of 295  
     creating part dynamically 295, 313  
 IVBFileDialog 251  
 IVBFlyText  
     bubble help 270, 344  
     flyOverHelpHandler 344, 345  
     response time 344  
     using info area 261, 270, 271  
 IVBLongPart 283  
 IVBStringPart 249, 311  
 IView 461  
 IViewPort 164, 165  
 IVSequence 140, 143, 317

## K

key identifier 293  
 key identifier. See data identifier under  
     Data Access Builder part  
 keyboard handler. See handler

## L

language 3  
 language 5–7, 10–15, 43, 83  
     support for NLS 30  
 legacy code 83, 226, 367  
 LIBPATH 123, 184

Linker 45  
linker. See ILINK  
locales  
    ICONV facility 42  
    IEEE POSIX P1003.2 42  
    LOCALDEF facility 42  
long part. See IVBLongPart  
Lotus Approach 402  
LPEX editor 40, 230

## M

makefile 136, 145  
MDX file 398, 405  
menu  
menu. See IMenuItem, IMenuSeparator  
module definition. See DEF file  
multicell canvas. See IMultiCellCanvas  
multimedia. See IMMDigitalVideo  
multiple line edit control. See IMultiLineEdit

## N

name mangling 446  
naming convention 122, 148, 366  
NDX file 398  
NetBIOS 377, 380  
nonvisual part  
    connecting to a visual part 249  
notification framework  
    dispatchNotificationEvent method 359, 371  
    enableNotification method 358  
    handleNotificationFor method 357, 371  
    notification event 354, 370, 371  
    notifier 226, 353, 354, 358, 361, 363, 369  
    notifyObservers method 232, 306, 358, 359, 362, 363, 369  
    observer 232, 353, 354, 359, 361, 370, 371  
notifier. See IStandardNotifier  
NULL 400, 402  
numeric spin button. See INumericSpinButton

## O

object  
    attribute 6, 8  
    business object 91, 225, 226  
    design 15, 81, 82, 87  
    finding 71  
    function 6, 7, 8, 10  
    interface 6, 7, 8, 10, 243, 354, 363, 364, 366, 368  
    semantic object 15  
    technical object 225  
Object Linking and Embedding. See OLE  
Object Management Group. See OMG  
Object Modeling Technique. See OMT  
object-oriented analysis. See OOA  
object-oriented design. See OOD  
object-oriented method. See OMT, OOSE, RDD, VMT  
Object-Oriented Software Engineering. See OOSE  
observer. See IObserver  
ODBC 385, 387, 389, 397, 403  
    access method 141  
    Administrator 389, 404  
    conformance level 386  
    Data Source 134, 386, 394, 404  
    Driver 386, 389, 397, 400, 403  
    Driver Manager 386, 387  
OLE  
    automation 457  
    client 456  
    CLSID 463  
    component 456  
    component stationery 458  
    compound files 460  
    container 456  
    data streaming 461  
    dispatchable interface 457  
    distributed 456  
    GUI bundle 458  
    IComponent 460  
    IComponentStationery 462  
    IComponentStationeryFor 462  
    IDocumentStorage 461  
    IEmbedderModel 459  
    IFlatFileStorage 461  
    IGUIBundle 463  
    IModel 459  
    in-place activation 455, 457  
    interface 456  
    IStructuredStorage 461  
    IView 461  
    metafile 457

- model 458
- moniker 457
- notification 456
- out-place activation 457
- root container component 459
- server 456
- structured storage 456, 460
- uniform data transfer 456
- view 458
- OLE Support 38
- OMG 422
- OMT
  - class dictionary 72, 73, 82, 521
  - dynamic model 12, 247
  - notation 511
  - static model 12
- OOA
  - building use case 66
  - class dictionary 82
  - deliverable 62, 80, 82
  - user interface prototype 62, 68, 69, 82, 88
- OOD
  - deliverable 82
  - object design 15, 81, 82, 87
  - system design 15, 81–83
- OOSE
  - use case 13
- Open Class Library
  - Application Support Class Library 37
  - Collection Class Library 29, 37
  - Collection Smart Guide 37
  - Data Access Builder Class Library 37
  - Standard Class Libraries 38, 273
  - User Interface Class Library 29, 35, 117
- Open Database Connectivity, See ODBC 385
- Oracle 7 387, 397

## P

- page 113
- part
  - abstract class 354
  - as observer 232, 353, 354, 359–361, 370, 371
  - asString method 202, 272, 414
  - attribute 226, 232, 245, 344
  - browsing 327
  - clone 296
  - constructor 32, 144, 295, 296
  - dictionary 521
  - feature source code 229, 367

- generating source code 229, 273, 283, 367
- import 140, 243, 244, 353, 366, 368
- initialization 358
- instance 266, 274, 295–299, 313, 316, 328, 339
- interface 25–26, 30, 33, 243, 354, 363, 364, 366, 368
- method 227–232, 273–284, 285, 291, 293, 295, 347, 366
- move 140
- naming convention 122, 148, 366
- prefix 273, 286, 311
- promoting a feature 31, 158, 162, 176, 217, 267, 270, 287, 346, 370
- saving 305
- tabbing and depth order 254
- Part Interface Editor
  - creating actions 31
  - creating attributes 30, 228
  - creating events 31
  - promoting features 31
  - selecting preferred features 30, 32
  - user-defined files 229, 274, 281, 283, 285, 305, 312
- password 135, 142, 375
- pattern 63, 70
- performance analyzer 52
  - pattern recognition 54
- persistent data 90, 106
- persistent object 139, 391
- persistent object. See IPersistentObject
- PM 93
- polymorphism 10, 83
- pop-up menu 254
- portability 35, 36, 42, 106, 108, 388
- pragma 137, 445, 446
- Presentation Manager. See PM
- primitive part 25, 104
- problem domain 12, 13, 14, 61, 63, 67, 72, 85
- profiler. See Performance analyzer
- profile. See IProfile
- project
  - action 21
  - catalog 22
  - composite 23
  - element 20
  - installation script 22
  - source 21
  - target 20
  - type 21
- promoted feature
- public interface 370
- push button. See IPushButton

---

## R

RDD 12, 13  
REAL database 134, 142, 376, 389, 515  
registry 463  
relationship  
    use 295, 313  
release-to-release binary compatibility  
    See SOM, RBBC  
requirement specification 62, 66, 71, 92, 346  
Responsability-Driven Design. See RDD  
reusability  
    design consideration 258  
    using variable part 259  
REXX 40, 368  
role 67–69

## S

screen resolution 152, 153, 154, 184, 212  
sequence. See IVSequence  
set canvas. See ISetCanvas  
Smalltalk 422, 424, 430  
SOM 33, 43, 44, 138  
    attribute 442  
    def emitter 433  
    environment structure 430  
    implementation binding 429  
    implementation template 429  
    language neutrality 422  
    metaclass 425, 437  
    RRBC 423, 428, 436  
    usage binding 429  
    xc emitter 441  
SOMClass 426  
SOMClassMgr 427  
SOMClassName 447  
SOMDataName 447  
somDefaultInit 441  
SOMLINK 429  
SOMMSingleInstance 426  
SOMMTraced 426, 437  
SOMNoMangling 447, 448  
SOMObject 426, 444  
somPrintSelf 441  
SOMReleaseOrder 449  
somSelf 429  
SOM\_Scope 429  
source code xxix, 33, 40, 44, 68, 91, 136, 203,  
    226, 227, 229, 363, 367, 383, 525  
specialization 9  
SQL 383

    clause 249, 291  
    dynamic 383  
    file 403, 409  
    select statement 408, 410  
    static 383  
SQL clause  
SQLPREP 112, 113  
SQX file 136, 392  
standard canvas. See ICanvas  
static model  
    aggregation 15  
    association 77, 78, 93, 100, 144, 160  
    link 71, 75, 77, 78, 93–97, 100, 295, 307,  
        313  
static text. See IStaticText  
Sticky mark 155, 156, 255  
string generator. See IStringGenerator  
string part. See IVBStringPart  
Structured Query Language. See SQL  
subsystem 20, 23, 46, 82, 83, 84, 88, 90, 92,  
    106  
Sybase System 10 387, 397  
System Object Model. See SOM  
system variable 347

## T

tabbing and depth order 254  
TCP/IP 378  
toolbox. See VBSAMPLES.VBB  
traceability 68, 87  
transaction management 225

## U

use case 12, 66–69  
user ID 135, 142, 375, 382  
User Interface Class Library

## V

variable part  
    and factory object 295  
    change type 394  
    tearing off an attribute 344  
    using 266  
VBB file 140  
VBBASE.VBB file 122  
VBDAX.VBB file 141, 150



---

- VBE file 136, 140, 368
- VBMM.VBB file 150, 171, 253
- VBSAMPLE.VBB file 150, 308, 311, 327
- view
  - hierarchy 149
- view port. See IViewPort
- view. See visual part
- Visual Builder
  - concepts 25–29
  - editors 29–33
  - invoke 141
  - SmartHouse example 26
- Visual Modeling Technique. See VMT
- Visual part
  - hierarchy 149
- Visual Realty application 57, 64, 67, 72, 81, 84, 86, 88, 89, 104, 123, 147, 218
- VisualAge for C++ 16, 87
- VisualAge Smalltalk 16
- VMT
  - roots 12
  - visual programming 13
- VRDacs.vbb 394, 411

## W

- window
  - modal 297, 298, 300, 328, 330, 339, 343
  - modeless 314, 317
- WorkFrame
  - Actions 125
  - Build Smarts facility 24, 120
  - concepts 20–24
  - integrated development environment 20
  - MakeMake facility 23, 115
  - Project Smarts facility 23, 109
  - Types 125
- WorkFrame Configuration File
  - contents 124
  - setup 123
- World Wide Web. See WWW
- wrapper 251, 326, 327
- WWW xxxiii

## X, Y, Z

- X/Open 385

## LICENSE AGREEMENT AND LIMITED WARRANTY

READ THE FOLLOWING TERMS AND CONDITIONS CAREFULLY BEFORE OPENING THIS CD PACKAGE. THIS LEGAL DOCUMENT IS AN AGREEMENT BETWEEN YOU AND PRENTICE-HALL, INC. (THE "COMPANY"). BY OPENING THIS SEALED CD PACKAGE, YOU ARE AGREEING TO BE BOUND BY THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THESE TERMS AND CONDITIONS, DO NOT OPEN THE CD PACKAGE. PROMPTLY RETURN THE UNOPENED CD PACKAGE AND ALL ACCOMPANYING ITEMS TO THE PLACE YOU OBTAINED THEM FOR A FULL REFUND OF ANY SUMS YOU HAVE PAID.

1. **GRANT OF LICENSE:** In consideration of your purchase of this book, and your agreement to abide by the terms and conditions of this Agreement, the Company grants to you a nonexclusive right to use and display the copy of the enclosed software program (hereinafter the "SOFTWARE") on a single computer (i.e., with a single CPU) at a single location so long as you comply with the terms of this Agreement. The Company reserves all rights not expressly granted to you under this Agreement.

2. **OWNERSHIP OF SOFTWARE:** You own only the magnetic or physical media (the enclosed CD) on which the SOFTWARE is recorded or fixed, but the Company and the software developers retain all the rights, title, and ownership to the SOFTWARE recorded on the original CD copy(ies) and all subsequent copies of the SOFTWARE, regardless of the form or media on which the original or other copies may exist. This license is not a sale of the original SOFTWARE or any copy to you.

3. **COPY RESTRICTIONS:** This SOFTWARE and the accompanying printed materials and user manual (the "Documentation") are the subject of copyright. The individual programs on the CD are copyrighted by the authors of each program. Some of the programs on the CD include separate licensing agreements. If you intend to use one of these programs, you must read and follow its accompanying license agreement. If you intend to use the trial version of Internet Chameleon, you must read and agree to the terms of the notice regarding fees on the back cover of this book. You may not copy the Documentation or the SOFTWARE, except that you may make a single copy of the SOFTWARE for backup or archival purposes only. You may be held legally responsible for any copying or copyright infringement which is caused or encouraged by your failure to abide by the terms of this restriction.

4. **USE RESTRICTIONS:** You may not network the SOFTWARE or otherwise use it on more than one computer or computer terminal at the same time. You may physically transfer the SOFTWARE from one computer to another provided that the SOFTWARE is used on only one computer at a time. You may not distribute copies of the SOFTWARE or Documentation to others. You may not reverse engineer, disassemble, decompile, modify, adapt, translate, or create derivative works based on the SOFTWARE or the Documentation without the prior written consent of the Company.

5. **TRANSFER RESTRICTIONS:** The enclosed SOFTWARE is licensed only to you and may not be transferred to any one else without the prior written consent of the Company. Any unauthorized transfer of the SOFTWARE shall result in the immediate termination of this Agreement.

6. **TERMINATION:** This license is effective until terminated. This license will terminate automatically without notice from the Company and become null and void if you fail to comply with any provisions or limitations of this license. Upon termination, you shall destroy the Documentation and all copies of the SOFTWARE. All provisions of this Agreement as to warranties, limitation of liability, remedies or damages, and our ownership rights shall survive termination.

7. **MISCELLANEOUS:** This Agreement shall be construed in accordance with the laws of the United States of America and the State of New York and shall benefit the Company, its affiliates, and assignees.

8. **LIMITED WARRANTY AND DISCLAIMER OF WARRANTY:** The Company warrants that the SOFTWARE, when properly used in accordance with the Documentation, will operate in substantial conformity with the description of the SOFTWARE set forth in the Documentation. The Company does not warrant that the SOFTWARE will meet your requirements or that the operation of the SOFTWARE will be uninterrupted or error-free. The Company warrants that the media on which the SOFTWARE is delivered shall be free from defects in materials and workmanship under normal use for a period of thirty (30) days from the date of your purchase. Your only remedy and the Company's only obligation under these limited warranties is, at the Company's option, return of the warranted item for a refund of any amounts paid by you or replacement of the item. Any replacement of SOFTWARE or media under the warranties shall not extend the original warranty period. The limited warranty set forth above shall not apply to any SOFTWARE which the Company determines in good faith has been subject to misuse, neglect, improper installation, repair, alteration, or damage by you. EXCEPT FOR THE EXPRESSED WARRANTIES SET FORTH ABOVE, THE COMPANY DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. EXCEPT FOR THE EXPRESS WARRANTY SET FORTH ABOVE, THE COMPANY DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATION REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE.

IN NO EVENT, SHALL THE COMPANY OR ITS EMPLOYEES, AGENTS, SUPPLIERS, OR CONTRACTORS BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR IN CONNECTION WITH THE LICENSE GRANTED UNDER THIS AGREEMENT, OR FOR LOSS OF USE, LOSS OF DATA, LOSS OF INCOME OR PROFIT, OR OTHER LOSSES, SUSTAINED AS A RESULT OF INJURY TO ANY PERSON, OR LOSS OF OR DAMAGE TO PROPERTY, OR CLAIMS OF THIRD PARTIES, EVEN IF THE COMPANY OR AN AUTHORIZED REPRESENTATIVE OF THE COMPANY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL LIABILITY OF THE COMPANY FOR DAMAGES WITH RESPECT TO THE SOFTWARE EXCEED THE AMOUNTS ACTUALLY PAID BY YOU, IF ANY, FOR THE SOFTWARE.

SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT ALWAYS APPLY. THE WARRANTIES IN THIS AGREEMENT GIVE YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY IN ACCORDANCE WITH LOCAL LAW.

#### ACKNOWLEDGMENT

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU ALSO AGREE THAT THIS AGREEMENT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN YOU AND THE COMPANY AND SUPERSEDES ALL PROPOSALS OR PRIOR AGREEMENTS, ORAL, OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN YOU AND THE COMPANY OR ANY REPRESENTATIVE OF THE COMPANY RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Should you have any questions concerning this Agreement or if you wish to contact the Company for any reason, please contact in writing at the address below.

Robin Short  
Prentice Hall PTR  
One Lake Street  
Upper Saddle River, New Jersey 07458

## **LICENSE INFORMATION FOR DB2 Trial Version**

The products on this CD-ROM are licensed under the International Program License Agreement (IPLA) terms and conditions.

The following products entitle one person to use the product:

- DB2 for Windows 95 & Windows NT Single-User,
- DB2 Software Developer's Kit for Windows 95 & Windows NT
- DDCS for Windows NT Single-User.

The DB2 Server product and the DDCS Multi-User Gateway product entitle up to five concurrent users. If more than five people need to use these products concurrently, you must purchase Entitlements for Additional Users. These Entitlements are available for 1, 5, 10, or 50 additional users.

This program has one or more features that are designed for use on machines other than the machine on which the Program is used. You may make copies of the feature(s) and documentation in support of your authorized use of the Program. Persons using a machine outside of your Enterprise may use the copy only to access the associated Program.

You can install and try any of the DB2 or DDCS products (other than the product that you bought) for a period of 60 days or until July 31, 1998 whichever comes first. After the trial period expires, the products become inactive until a license key is entered into the nodelock file. If you decide you want to buy a product you tried, contact your IBM software reseller or the IBM office nearest you.

When you purchase one of the DB2 or DDCS CD-ROM products, you receive a label which contains a license key. These labels are not transferable and should be kept in a secure place. To permanently install the product, you enter the license key in the nodelock file on the machine where the product is installed. You can enter the license key at installation time or you can install the product and enter the key in the nodelock file at a later time.

If you buy a product after the trial period has ended, you do not need to re-install the product. Instead, you would edit the nodelock file and add the license key that you received in the product box to reactivate the product.

## **LICENCE AGREEMENT FOR VisualAge for C++ for Windows Trial Version**

This is a no charge Evaluation License ("License") between you and International Business Machines Corporation ("IBM") for the evaluation of IBM's software and related documentation. ("Program")

IBM grants you a non-exclusive, non-transferable license to the Program only to enable you to evaluate the potential usefulness of the Program to you. You may not use the Program for any other purpose and you may not distribute any part of it, either alone or with any of your software products.

IBM retains ownership of the Program and any copies you make of it. You may use the Program on one (1) machine only.

You may not decompile, disassemble or otherwise attempt to translate or seek to gain access to the Program's source code.

The term of your License will be from the date of first installation of the Program, and will terminate 60 days later, unless otherwise specified. THE PROGRAM WILL STOP FUNCTIONING WHEN THE LICENSE TERM EXPIRES. You should therefore take precautions to avoid any loss of data that might result. You must destroy and/or delete all copies you have made of the Program within ten (10) days of the expiry of your License.

If you are interested in continuing to use the Program after the end of your License, you must place an order for a full license to the Program and pay the applicable license fee. In that event, your use of the Program will be governed by the provisions of the applicable IBM license for the Program.

IBM accepts no liability for damages you may suffer as a result of your use of the Program. In no event will IBM be liable for any indirect, special or consequential damages, even if IBM has been advised of the possibility of their occurrence.

YOU UNDERSTAND THAT THE PROGRAM IS BEING PROVIDED TO YOU "AS IS," WITHOUT ANY WARRANTIES (EXPRESS OR IMPLIED) WHATSOEVER, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, PERFORMANCE OR FITNESS FOR ANY PARTICULAR PURPOSE. Some jurisdictions do not allow the exclusion or limitation of warranties or consequential or incidental damages, so the above may not apply to you.

IBM may terminate your License at any time if you are in breach of any of its terms.

This License will be governed by and interpreted in accordance with the laws of the State of New York.

This License is the only understanding and agreement we have for your use of the Program. It supercedes all other communications, understandings or agreements we may have had prior to this License.

## **DB2 for Windows NT and Windows 95**

### **Hardware requirements**

Intel 486 or higher processor

For Clients, at least 1 MB of memory in addition to the OS requirements

For Servers, 10 MB or more RAM.

Hard disk space requirements:

DB2 CAE: 9 MB

DB2 SDK: 17 MB

### **Software Requirements**

Windows NT 3.51 or higher

Windows 95

### **Communications Protocols**

IPX/SPX or NetBIOS or TCP/IP or APPC or local IPC

# VisualAge for C++ for Windows

## Hardware Requirements

- Processor:
    - 80386 minimum
    - 80486 or higher strongly recommended
  - Display:
    - VGA minimum
    - SVGA recommended
  - Mouse or pointing device
  - CD-ROM drive required\
  - Memory requirements for Win95 (add 4 MB for Win NT):
    - C development: 8 MB minimum, 12 MB recommended
    - C++ development: 12 MB minimum, 16 MB recommended
    - C++ visual development: 16 MB minimum, 24 MB recommended
  - Disk space requirements:
    - Minimal when running product from the CD-ROM
- Custom install of the following is also available:
- 285 MB for all tools and toolkits
  - 60 MB for samples and tutorials
  - 25 MB for all documentation
  - 40 MB swap space minimum

## Software Requirements

- Development environment:
  - Windows NT 3.51 or higher
  - Windows 95
- Execution environment:
  - Windows 3.1
  - Windows NT 3.51 or higher
  - Windows 95

## Prereq products

- For Data Access Builder
  - DB2 for Windows NT Version 2.1 and higher, or
  - Oracle 7 and higher, or
  - Sysbase SQL Server 10 and higher.







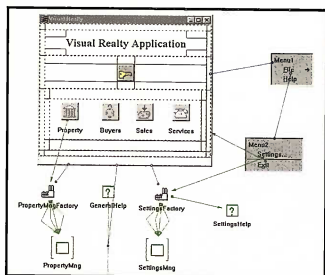
VisualAge for C++  
Object-Oriented  
Associate  
Developer



VisualAge for C++  
Object-Oriented  
Developer

# Programming with VisualAge for C++ for Windows

Marc Carrel-Billiard • Michael Friess • Isabelle Mauny



## Learn VisualAge for C++ by example

*Programming with VisualAge for C++ for Windows* guides you through the development of a full-function application, from specifying requirements to implementation, using today's object orientation and visual programming.

Because this book focuses on the practice rather than the theory of object technology, you can use it to build an actual application, including relational database support, video and sound capability, and numerous graphical controls for a truly intuitive graphical user interface.

### *Programming with VisualAge for C++ for Windows* has four parts and includes a CD-ROM:

- Part 1 introduces the concepts and terms of visual programming and object-oriented techniques.
- Part 2 examines the analysis and design of static and dynamic models of the application.
- Part 3 shows how to use the tools of VisualAge for C++ to develop the sample application.
- Part 4 provides more details to enhance your skills in applying VisualAge for C++.
- The CD-ROM holds the evaluation version of VisualAge for C++ for Windows, the evaluation version of DB2, and the sample application.

Learn how to exploit the newest trends in software engineering. With the techniques in this book, Windows application construction has never been easier.

### About the Authors

MARC CARREL-BILLIARD, from IBM France, works at the IBM International Technical Support Organization in San Jose, California.

MICHAEL FRIESS works in Stuttgart for the Developer Support Organization of IBM Germany.

ISABELLE MAUNY works in La Gaude, France, for IBM EMEA Software Technical Support.

### ► Sharing Technical Expertise from around the World

This book and other IBM Redbooks are products of IBM's International Technical Support Organization, where worldwide specialists work alongside you to harness IBM technologies. IBM Redbooks make the answers to your most pressing technical questions easily and immediately accessible.

For more information: <http://www.redbooks.ibm.com/redbooks>

PRENTICE HALL  
Upper Saddle River, NJ 07458

<http://www.prenhall.com>

U.S. \$49.95/Canada \$69.95

5624-4782-00



PRINTED IN USA



0 76092 00179 9

ISBN 0-13-618208-9



9 780136 182085